Running Csound in Parallel
LAC2009

John ffitch

University of Bath

LAC Maynooth, May 2011

Introduction

We are seeing more processors rather than faster ones.

The challenge now is to find ways of using multiple cores effectively to improve the performance of a single program.

NB: No interest here in efficiency

Historical Note

I first stated this in the early 1970s, and at intervals since, but now the need is much more imminent!

The Hardware Imperative

Robert S. Barton, one the greats of early computing, said there is a *technological imperative*; what hardware requires is a forcing term on software.

Attempts at parllelism at that time did not succeed, but we can learn from the experience.

The Hardware Imperative

Robert S. Barton, one the greats of early computing, said there is a *technological imperative*; what hardware requires is a forcing term on software.

Attempts at parllelism at that time did not succeed, but we can learn from the experience.

A Brief Biased History of Parallelism

Forty years ago I was proposing a parallel functional machine, and thirty years ago we built to Bath Concurrent LISP Machine, a cluster of six M68000 processors with each processor having three shared memory windows with one other.

Twenty years ago we built the a LISP-based Concurrent Object-Oriented system.

A Brief Biased History of Parallelism

Forty years ago I was proposing a parallel functional machine, and thirty years ago we built to Bath Concurrent LISP Machine, a cluster of six M68000 processors with each processor having three shared memory windows with one other.

Twenty years ago we built the a LISP-based Concurrent Object-Oriented system.

Concurrent Software

We based our work on the premis that users cannot be expected (or trusted) to modify their thinking for parallel execution, and the responsibility needs to be taken by the software translation system that converts the program or specification into an executable form.

Compiler analysis can be extended to inform the structure; described variously in PhD Thesis of Marti (1980), and papers by me in computer algebra.

Concurrent Software

We based our work on the premis that users cannot be expected (or trusted) to modify their thinking for parallel execution, and the responsibility needs to be taken by the software translation system that converts the program or specification into an executable form.

Compiler analysis can be extended to inform the structure; described variously in PhD Thesis of Marti (1980), and papers by me in computer algebra.

The Two Critical Points

A: Two entities can be run at the same time if they do not reference/modify shared data

B: Two entities should be run at the same time if the overhead is less than the gain

The Two Critical Points

A: Two entities can be run at the same time if they do not reference/modify shared data

B: Two entities should be run at the same time if the overhead is less than the gain

Ab Initio Parallelism?

Should we just start coding again? I say not as there is too much already committed.

Two attempts however are worth mentioning;

- Csound in real-time using Transputers
- Midas streamed DSP network

Both are finer-grained than what I am advocating

Ab Initio Parallelism?

Should we just start coding again? I say not as there is too much already committed.

Two attempts however are worth mentioning;

- Csound in real-time using Transputers
- Midas streamed DSP network

Both are finer-grained than what I am advocating

Towards a Parallel Csound

Csound has been in existence and development for 25 years. It provides **instruments** that are played following a score.

The instruments are activated, performed and deactivated using a control cycle (running at a control rate). Instruments are performed in a defined order, and so interaction between instruments has defined behaviour.

```
until end of events do
  deal with notes ending
  sort new events onto instance list
  for each instrument in instance list
     calculate instrument
```

Towards a Parallel Csound

Csound has been in existence and development for 25 years. It provides **instruments** that are played following a score.

The instruments are activated, performed and deactivated using a control cycle (running at a control rate). Instruments are performed in a defined order, and so interaction between instruments has defined behaviour.

```
until end of events do
  deal with notes ending
  sort new events onto instance list
  for each instrument in instance list
     calculate instrument
```

Towards a Parallel Csound (b)

Making this parallel could be to make the loop parallel, as long as there is no interaction, so....

- Following Marti we can use code analysis techniques
- Only global variables matter.
- For each instrument determine the sets of globals are read, written, or both
- Use this to control the loop

Towards a Parallel Csound (b)

Making this parallel could be to make the loop parallel, as long as there is no interaction, so....

- Following Marti we can use code analysis techniques
- Only global variables matter.
- For each instrument determine the sets of globals are read, written, or both
- Use this to control the loop

Towards a Parallel Csound (b)

Making this parallel could be to make the loop parallel, as long as there is no interaction, so....

- Following Marti we can use code analysis techniques
- Only global variables matter.
- For each instrument determine the sets of globals are read, written, or both
- Use this to control the loop

Towards a Parallel Csound (b)

Making this parallel could be to make the loop parallel, as long as there is no interaction, so....

- Following Marti we can use code analysis techniques
- Only global variables matter.
- For each instrument determine the sets of globals are read, written, or both
- Use this to control the loop

Towards a Parallel Csound (b)

Making this parallel could be to make the loop parallel, as long as there is no interaction, so....

- Following Marti we can use code analysis techniques
- Only global variables matter.
- For each instrument determine the sets of globals are read, written, or both
- Use this to control the loop

Towards a Parallel Csound (c)

Special case: most instruments add into the output bus, but this is not an operation that needs ordering (subject to rounding errors), although it may need a mutex or spin-lock. The language processing can insert any necessary protections in these cases.

There are other globals than variables but the idea is the same.

Towards a Parallel Csound (c)

Special case: most instruments add into the output bus, but this is not an operation that needs ordering (subject to rounding errors), although it may need a mutex or spin-lock. The language processing can insert any necessary protections in these cases.

There are other globals than variables but the idea is the same.

Design

Build a DAG of ordering dependancy, where the arcs represent the need to be evaluated before the descendents

```
until end of events do
  deal with notes ending
  add new events and reconstruct the DAG
  until DAG empty
    foreach processor
      evaluate a root from DAG
  wait until all processes finish
```

Design

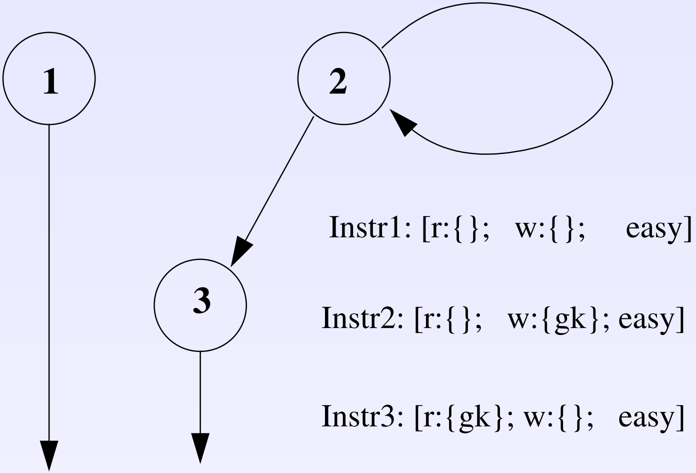
Build a DAG of ordering dependancy, where the arcs represent the need to be evaluated before the descendents

```
until end of events do
  deal with notes ending
  add new events and reconstruct the DAG
  until DAG empty
    foreach processor
      evaluate a root from DAG
  wait until all processes finish
```

Compiler Example

Using the new parser the information is “easily” gathered and the bus-locks inserted.

```
instr 1
  a1 oscil p4, p5, 1
     out   a1
endin
instr 2
  gk oscil p4, p5, 1
endin
instr 3
 a1  oscil gk, p5, 1
      out  a1
endin
```


1
2
3
Instr1: [r:{ }; w:{ }; easy]
Instr2: [r:{ }; w:{gk}; easy]
Instr3: [r:{gk}; w:{ }; easy]

Maintaining the DAG

This is a major problem. It is consumed on each cycle, but adding and losing instances means DAG must be remade, not just copied. The current version of representation and algorithm is the result of much experimentation and probably could be improved.

Locking and Barriers

We use the POSIX pthreads library.

- One master thread does analysis and DAG construction
- A Barrier at the start of each control cycle
- Each worker gets a task from the DAG, with a mutex
- At end of instrument-cycle DAG is modified
- When no work proceed to end Barrier

Locking and Barriers

We use the POSIX pthreads library.

- One master thread does analysis and DAG construction
- A Barrier at the start of each control cycle
- Each worker gets a task from the DAG, with a mutex
- At end of instrument-cycle DAG is modified
- When no work proceed to end Barrier

Locking and Barriers

We use the POSIX pthreads library.

- One master thread does analysis and DAG construction
- A Barrier at the start of each control cycle
- Each worker gets a task from the DAG, with a mutex
- At end of instrument-cycle DAG is modified
- When no work proceed to end Barrier

Locking and Barriers

We use the POSIX pthreads library.

- One master thread does analysis and DAG construction
- A Barrier at the start of each control cycle
- Each worker gets a task from the DAG, with a mutex
- At end of instrument-cycle DAG is modified
- When no work proceed to end Barrier

Locking and Barriers

We use the POSIX pthreads library.

- One master thread does analysis and DAG construction
- A Barrier at the start of each control cycle
- Each worker gets a task from the DAG, with a mutex
- At end of instrument-cycle DAG is modified
- When no work proceed to end Barrier

Locking and Barriers

We use the POSIX pthreads library.

- One master thread does analysis and DAG construction
- A Barrier at the start of each control cycle
- Each worker gets a task from the DAG, with a mutex
- At end of instrument-cycle DAG is modified
- When no work proceed to end Barrier

Load Balancing

We would like each task to be equal computation and sufficiently large.

This is not always true and currently we ignore this problem

Code exists to collect instances together.

Load Balancing

We would like each task to be equal computation and sufficiently large.

This is not always true and currently we ignore this problem

Code exists to collect instances together.

Load Balancing

We would like each task to be equal computation and sufficiently large.

This is not always true and currently we ignore this problem

Code exists to collect instances together.

Load Data

We are collecting data on average instruction count for opcodes, using valgrind.

We calculate three counts; initialisation, per k-cycle, per sample

Costs of a few opcodes

Opcode	init	Audio	Control
table.a	93	23.063	43.998
table.k	93	0	45
butterlp	9	29.005 4	5.478
butterhi	19	30.000	35
butterbp	20	30	71
bilbar	371.5	1856.028	86
ags	497	917.921	79475.155
oscil.kk	69	12	47
oscili.kk	69	21	49
reverb	6963.5	77	158

Current State

Implemented by Chris Wilson, revised by John ffitch. Tested on Linux (and OSX). Requires the new parser but is available on Sourceforge as a branch. Can control number of threads.

Some features still missing, like zak and buses.

Current State

Implemented by Chris Wilson, revised by John ffitch. Tested on Linux (and OSX). Requires the new parser but is available on Sourceforge as a branch. Can control number of threads.

Some features still missing, like zak and buses.

Linux Quadcore Results

Sound	ksmps	1			5	Time
Xanadu	1	**31.202**	39.291	42.318	43.043	48.304
Xanadu	10	**18.836**	19.901	20.289	21.386	22.485
Xanadu	100	16.023	17.413	16.999	16.545	**15.884**
Xanadu	300	17.159	16.137	15.141	15.723	**14.905**
Xanadu	900	16.004	15.099	**13.778**	14.364	14.167
CloudStrata	1	**173.757**	191.421	211.295	214.516	261.238
CloudStrata	10	89.406	**80.998**	94.023	110.170	98.187
CloudStrata	100	85.966	86.114	**81.909**	83.258	85.631
CloudStrata	300	87.153	76.045	79.353	78.399	**74.684**
CloudStrata	900	82.612	76.434	**64.368**	76.217	74.747
trapped	1	**20.931**	63.492	81.654	107.982	139.334
trapped	10	**3.348**	7.724	9.500	12.165	14.937
trapped	100	**1.388**	1.810	1.928	2.167	2.612
trapped	300	1.319	**1.181**	1.205	1.386	1.403
trapped	900	1.236	**1.025**	1.085	1.091	1.112

Performance

As the control rate decreases, corresponding to an increase in ksmps, the potential gain increases. This suggests that the current system is using too small a granularity and the collecting of instruments into larger groups will give a performance gain.

The performance figures are perhaps a little disappointing, but they do show that it is possible to get speed improvements, and more work on the load balance could be useful.

Performance

As the control rate decreases, corresponding to an increase in ksmps, the potential gain increases. This suggests that the current system is using too small a granularity and the collecting of instruments into larger groups will give a performance gain.

The performance figures are perhaps a little disappointing, but they do show that it is possible to get speed improvements, and more work on the load balance could be useful.

Conclusions

A system for parallel execution of Csound has been presented, that works at the granularity of the instrument, based on thirty-year old technology.

I believe that the level of granularity is the correct one, and with more attention to the DAG construction and load balancing it offers real gains for many users. It does not require specialist hardware, and can make use of current and projected commodity systems.

Conclusions

A system for parallel execution of Csound has been presented, that works at the granularity of the instrument, based on thirty-year old technology.

I believe that the level of granularity is the correct one, and with more attention to the DAG construction and load balancing it offers real gains for many users. It does not require specialist hardware, and can make use of current and projected commodity systems.

Acknowledgements

- Jed Marti (ex U of Utah and RAND; ARTIS, LLC)
- Arthur Norman (Trinity Collge, Cambridge)
- Chris Wilson (ex U of Bath; Imagination Technology plc)
- Steven Yi
- Csound Community

Thanks to Codemist Ltd. This work in unsupported by public agencies