Citation for published version:
ffitch, J 2011, 'Running Csound in parallel' Paper presented at Linux Audio Conference 2011, National University of Ireland, Maynooth, 6/05/11 - 8/05/11, pp. 17-22.

Publication date:
2011

Link to publication

Running Csound in Parallel

John ffitch
Department of Computer Science
University of Bath
Bath BA2 7AY
United Kingdom
jpff@cs.bath.ac.uk

Abstract

One of the largest challenges facing computer scientists is how to harness multi-core processors into coherent and useful tools. We consider one approach to shared-memory parallelism, based on thirty year old ideas from the LISP community, and describe its application to one "legacy" audio programming system, Csound. The paper concludes with an assessment of the current state of implementation.

Parallelism, HiPAC, Csound

In the history of computing we have already seen rather often a mismatch between the available hardware and the state of software development. The current incarnation is currently as bad as it ever has been. While Moore's Law on the number of transistors on a chip still seems to be accurate, the commonly believed corollary, that processors get faster in the dame way, has been shown not to be the case.

Instead 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. This paper addresses this issue from a historical perspective and show how 1980s technology can be used, in particular to providing a faster Csound[Boulanger, 2000].

1 The Hardware Imperative

Computing has always had a hardware and a software aspect. It may be usual to view this as a harmonious marriage, but in reality there are a number of tensions in the relationship. Sometimes these conflicts are positive and stimulate innovation, so example the major improvements in compilation following RISC technology.

Usually software follows the hardware, driven by the *technological imperative* in the words of Robert S. Barton. When I worked with him the late 1970s it was also parallelism that was the cause, as we struggled to provide software to control massively parallel functional computers. I believe that there are many lessons to learn from that attempt to develop parallelism into a usable structure.

2 A Brief History of Parallelism

...and a biased one. Most of my involvement with parallelism has come from a functional or LISP style. For example we proposed a parallel functional machine forty years ago, but this widened following the Barton machine to more LISP-based systems, such as the Bath Concurrent LISP Machine [Fitch and Marti, 1984], and later the developments of simulation to object-based parallelism [Padget et al., 1991]. Much of this work is based on the thesis 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. In particular the compiler analysis can be extended to inform the structure. The particular LISP form of this was described by Marti [1980b; 1980a], and advocated in [Fitch, 1989b] and [Fitch, 1989a]. At the heart of this methodology is determining when different elements of a program (function or object-method) do not interact with each other.

The other aspect of parallelism that needs to be considered is not just **if** two entities can be run at the same time, but is it worthwhile. All too frequently the overheads of setting up the parallel section is greater that the benefit. The problem is in the general case, to know the cost of a computation is do the computation. This has led a number of compilation systems that perform testing runs of the program in order to estimate the performance. An alternative is to make a compile time estimate [Fitch and Marti, 1989]. Later, in section 5.5, we will make some use of both these techniques.

Parallelism has been an issue in computing for many years, and seems to re-emerge every twenty years as important. It is contended that we need to be mindful of what worked and what did not (and why) from the past.

3 Ab Initio Parallelism

Considering just the area of audio processing there is a body of existing code, albeit synthesis tools, analysis, mastering etc.. The obvious alternative to adapting these to a parallel machine would be to start again, and redesign the whole corpus with an eye on parallelism *ab initio*. The problem with this approach is the volume of software, and the commitment by users to these programs. The field of computer music has already suffered from software loss without inducing a whole new wave. For this reason the work advocated here has the preservation of the syntax and semantics of existing systems at its heart. This is indeed in line with the longstanding policy of Csound, never to break existing pieces.

Similarly dependence on user annotation is not the way forward. Skilled programmers are not noted for being good at the use of annotations, and we really should not expect our users, most of whom are musicians rather than programmers, to take this responsibility.

It should however be recognised that there have been attempts to recreate audio processing in parallel. Notably there was the 170 Transputer system that extended Csound into real-time [Bailey et al., 1990], which had hardware related problems of heat. A different approach was taken in [Kirk and Hunt, 1996] which streamed data through a distributed network of DSP processing units, to create Midas. Both of these have finer-grained distribution that the system presented here.

4 High Performance Computing

The mainstream projects in parallel processing are currently focused on HPC (High Performance Computing) which has come to mean matrix operations, using systems like MPI [Gropp et al., 1996]. The major interest is in partitioning of the matrix in suitable sizes for cache sizes, distribution between multicores and packet sizes for non-shared memories. Most of this is not directly applicable in audio processing, where low latency is such an important requirement.

This mismatch led to the promotion of **High Performance Audio Computing** in [Dobson et al., 2008], to draw attention to the differences, and in particular the latency. The other point about which I am concerned is that most of our users have commodity computers, usually with two or more cores, but not a cluster. The parallelism attempt in this paper is for the majority community rather than the privileged HPC users.

5 Towards a Parallel Csound

Csound [Vercoe, 1993] has a long and venerable history. It was written in the 1980s, and despite a significant rewrite ten years ago it remains grounded in the programming style of that period. As a member of the Music V family the system separates the *orchestra* from the *score*; that is it distinguishes the description of the sound from the sequence of notes. It also has a control rate, usually slower than the sampling rate, at which new events start, old one finish or control signals are sensed. Instruments are numbered by integers[1], and these labels play an important part in the Csound semantics. During any control cycle the individual instrument instances are calculated in increasing numerical order. Thus if one instrument is controlling another one, it will control the current cycle if it is lower numbered than the target, or the next cycle if higher. The main control loop can be described as

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

In order to introduce parallelism into this program the simplest suggestion is to make the "for each" loop run instances in parallel. If the instruments are truly independent then this should work, but if they interact in any way then the results may be wrong.

This is essentially the same problem that Marti tackled in his thesis. We can use code analysis techniques to determine which instruments are independent. Concentrating initially on variables, it is only global variables that are of concern. We can determine for each instrument the sets of global variables that are read, written, or both read and written, the last case corresponding to sole use, while many can read a variable as long as it is not written.

There is a special case which needs to be considered; most instruments add into the output

[1]They can be named, but the names are mapped to integers

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.

This thus gives a global design.

5.1 Design

The components in the design of parallel Csound are first a language analysis phase that can determine the non-local environment of each instrument specification. This is then used to organise the instance list into a DAG, where the arcs represent the need to be evaluated before the descendents. Then the main control operation becomes

```
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
```

We now consider the components of this.

5.2 Compiler

The orchestra language of Csound is basically simple, rather like an assembler with the operations being a range of DSP functions. The language processing in the usual Csound is simple, with a simple *ad hoc* lexer and hand-written parsing. It was a wish of the Csound5 re-write to produce a new parser, based on flex/bison, so things like functions with more than one argument could be introduced. A draft such parser was in construction while the major changes were made, as described in [ffitch, 2005]. The needs of parallelism added impetus to the new parser project, and it was largely completed by Yi, and is currently being subjected to extreme testing. The new parser was extended to construct the dependency information, and to add necessary locks (see section 5.4).

A simple example of the analysis for a simple orchestra (figure 1) can be seen in figure 2, listing the variables read, written and exclusive. The additional field is to indicate when the analysis has to assume that it might read or write anything.. In our simple example instrument 1 is independent of both instruments 2 and 3 (apart from the `out` opcode. On the other hand instrument 2 must run to completion before instrument 3, as it gives a value to a global read by instrument 3. Any number of instrument 3 instances can run at the same time but instances of instrument 2 need some care, as we must maintain the same order as a single threaded system.

This dependency analysis is maintained, and used in the DAG.

5.3 DAG

In the main loop to determine the execution of the instrument instances the decisions are determined by maintaining a DAG, the roots of which are the instruments that are available. In the case of our example the raw picture this is shown in figure 3. This DAG is consumed on each control cycle. Naïvely one must retain the original structure before consumption as it will be needed on the next cycle. This is complicated by the addition and deletion of notes. We investigated DAG updating algorithms but dynamic graphs is a complex area [Holm et al., 2001] and we are led to reject the allure of $O(\log(\log(n))$ algorithms; this complexity led us instead to a recreation of the DAG when there are changes. This is a summary of many experiments, and is one of the major bottlenecks in the system.

The whole dispatcher is very similar to a instruction scheduling algorithm such as [Muchnick and Gibbons, 2004] augmented by some VLIW concepts; it is in effect a bin-packing problem.

5.4 Locking and Barriers

The actually parallel execution is achieved with the POSIX pthreads library. One thread is designated as the main thread, and it is is that one that does the analysis and setup. There is a barrier set at the start of each control cycle so after the setup all threads are equal and try to get work from the DAG. This is controlled by a mutex so as not to compromise the structure. When an instrument-cycle finishes

```
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
```

Figure 1: A simple Orchestra.

```
Instr1: [r:{};   w:{};   easy]
Instr2: [r:{};   w:{gk}; easy]
Instr3: [r:{gk}; w:{};   easy]
```

Figure 2: Analysis of simple orchestra.

there is a further entry to the DAG via a mutex to remove the task and possibly release others. When there is no work the threads proceed to the barrier at the end. The master thread reasserts itself to prepare the next cycle. The mutex can be either POSIX mutexs or spinlocks, and we have experimented with both.

The other use of mutex/locks is in global variable updating. If a variable is added into, with a statement like

gk1 = gk1 + 1

then there is no need for exclusive use of the variable except during the updating. The compiler creates locks for each such occurrence and introduces calls to private opcodes (not available to users) to take and release the lock. There are other similar types of use that are not yet under the compiler control but could be (see section 5.6).

5.5 Load Balancing

A major problem in any non-synchronous parallel execution system is balancing the load between the active processes. Ideally we would like the load to be equal but this is not always possible. Also if the granularity of the tasks is too small then the overhead of starting and stopping a thread dominates the useful work. The initial system assumes that all instruments take about the same time, and that time is much larger than the setup time.

There is code written and tested but not yet

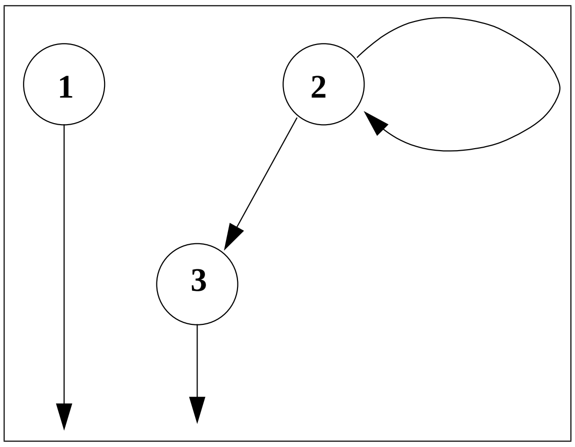

Figure 3: Partial Ordering from Analysis.

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

Table 1: Costs of a few opcodes.

deployed to collect instances together to ensure larger granularity. This needs raw data as to the costs of the individual unit generators. This data can come from static analysis(as in [Fitch and Marti, 1989]), or from running the program in a testing mode. In the case of Csound the basic generators are often constant in time, or we may assume some kind of average behaviour. We have been using valgrind on one system (actually Linux i386) to count instructions. With a little care we can separate the three components of cost; initialisation, instructions in each k-cycle and those obeyed on each audio sample. In the case of some of these opcodes the calculation do not take account of the time ranges due to data dependence, but we hope an average time is sufficient. These numbers, a small selection of which are shown in table 1, can be used for load balancing.

5.6 Current Status

The implementation of the above design, and many of its refinements are the work of Wilson[2009]. His initial implementation was on OSX and tested with a twin-core processor. The version currently visible on Sourceforge is a cleaned up version, with some of the experimental options removed and a more systematic use of mutexs and barriers.

The parser is enhanced to generate the dependency information and to insert small exclusion zones around global variable updates. The instrument dispatch loop has been rewritten along the lines in section 5, with the necessary DAG manipulations. There is code for load balancing but until the raw data is complete it is not deployed, but it has been tested.

Some opcodes, notably the **out** family have local spin locks, as they are in effect adding into a global variable. There are similar struc-

Sound	**ksmps**	1	2	3	4	5
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

Table 2: Performance figures; time in seconds.

tures in Csound that have not been suitably re-engineered, such as the **zak** global area and the software busses, which remain to be done.

The number of threads to be used is controlled by a command-line option. The design is not for massive parallelism, and the expectation is that the maximum number of threads will be about the same as the number of cores.

The limitations of the new parser, which is still being tested, and the missing locks and dependencies mean that the parallel version of Csound is not the main distributed one, but it is available for the adventurous.

6 Performance

All the above is of little point if there is no performance gain. It should be noted that we are concerned here with time to completion, and not overall efficiency. The need for parallelism here is to provide greater real-time performance and quicker composition.

The initial Wilson system reported modest gains on his dual core machine; 10% to 15% on a few examples with a top gain of 35%. The developed system has not seen such dramatic gains but they are there.

Running a range of tests on a Core-7 quad-core with hyper-threads it was possible to provide a wide range of results, varying the number of threads and the control rate. These are presented in figure 2 with the fastest time being in bold face. 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. It is clearly not always a winning strategy, but with the more complex scores there is a gain when ksmps is 100. Alternatively one might advise large values of ksmps, but that introduces quantisation issues and possibly zipped noise.

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.

7 Conclusions

A system for parallel execution of the "legacy" code in Csound has been presented, that works at the granularity of the instrument. The indications of overheads for this scheme suggest that we need to collect instruments into groups to increase granularity. The overall design, using compiler technology to identify the paces where parallelism cannot be deployed. The real cost of the system is in the recreation of the DAG and its consumption, and all too often this overhead swamps the gain from parallelism.

The remaining work that is needed before this can enter the main stream is partially the completion of the new parser, which is nearly done, and dealing with the other places in Csound where data is global. As well as the busses mentioned earlier there are global uses of tables. In the earlier versions of Csound tables were immutable, but recent changes has nullified this. The load balancing data needs to be collected. Currently this is a tedious process with much human intervention, and it needs to be scripted, not only to create the initial state but to make adding new opcodes into the parallel version.

Despite the problems identified in this paper parallel Csound is possible via this methodology. 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.

8 Acknowledgements

My thanks go to the many people who have contributed to the work here, In particular Jed Marti for the initial ideas, Arthur Norman for years of discussions, Steven Yi for the new parser and Chris Wilson for bringing it to reality; and the Csound developer community who encouraged me to continue.

References

N. Bailey, A. Purvis, P.D. Manning, and I. Bowler. 1990. Concurrent csound: Parallel execution for high-speed direct synthesis. In *Proceedings ICMC90*, Glasgow. ICMA.

Richard Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February.

Richard Dobson, John ffitch, and Russell Bradford. 2008. High Performance Audio Computing – A Position Paper. In *Proceedings of the 2008 ICMC*, pages 213–216, SARC, Belfast. ICMA and SARC.

John ffitch. 2005. The Design of Csound5. In *LAC2005*, pages 37–41, Karlsruhe, Germany, April. Zentrum für Kunst und Medientechnologie.

J. P. Fitch and J. B. Marti. 1984. The Bath Concurrent LISP machine. In *Proceedings of EUROCAL 1983*, volume 162 of *Lecture Notes in Computer Science*, pages 78–90.

J. P. Fitch and J. B. Marti. 1989. The static estimation of runtime. Technical Report 89–18, University of Bath Computing Group.

J. P. Fitch. 1989a. Can REDUCE be run in parallel? In *Proceedings of ISSAC89, Portland, Oregon*, pages 155–162. SIGSAM, ACM, July.

J. P. Fitch. 1989b. Compiling for parallelism. In J. Della-Dora and J. P. Fitch, editors, *Parallelism and Computer Algebra*, pages 19–32. Academic Press.

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22:789–828, September.

Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760.

Ross Kirk and Andy Hunt. 1996. MIDAS–MILAN: An Open Distributed Processing System for Audio Signal Processing. *J. Audio Eng. Soc.*, 44(3):119–129, March.

Jed B. Marti. 1980a. Compilation techniques for a control-flow concurrent lisp system. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 203–207, New York, NY, USA. ACM.

Jed B. Marti. 1980b. *A concurrent processing system for LISP*. Ph.D. thesis, University of Utah, Salt Lake City.

Steven S. Muchnick and Phillip B. Gibbons. 2004. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 39(4):167–174.

J. A. Padget, R. Bradford, and J. P. Fitch. 1991. Concurrent object-oriented programming in LISP. *Computer Journal*, 34:311–319.

Barry Vercoe, 1993. *Csound — A Manual for the Audio Processing System and Supporting Programs with Tutorials*. Media Lab, M.I.T.

Christopher Wilson. 2009. Csound Parallelism. Technical Report CSBU-2009-07, Department of Computer Science, University of Bath.