Challenging Multi-cores

Paul Cockshott[1]

[1]School of Computer Science

Motivation

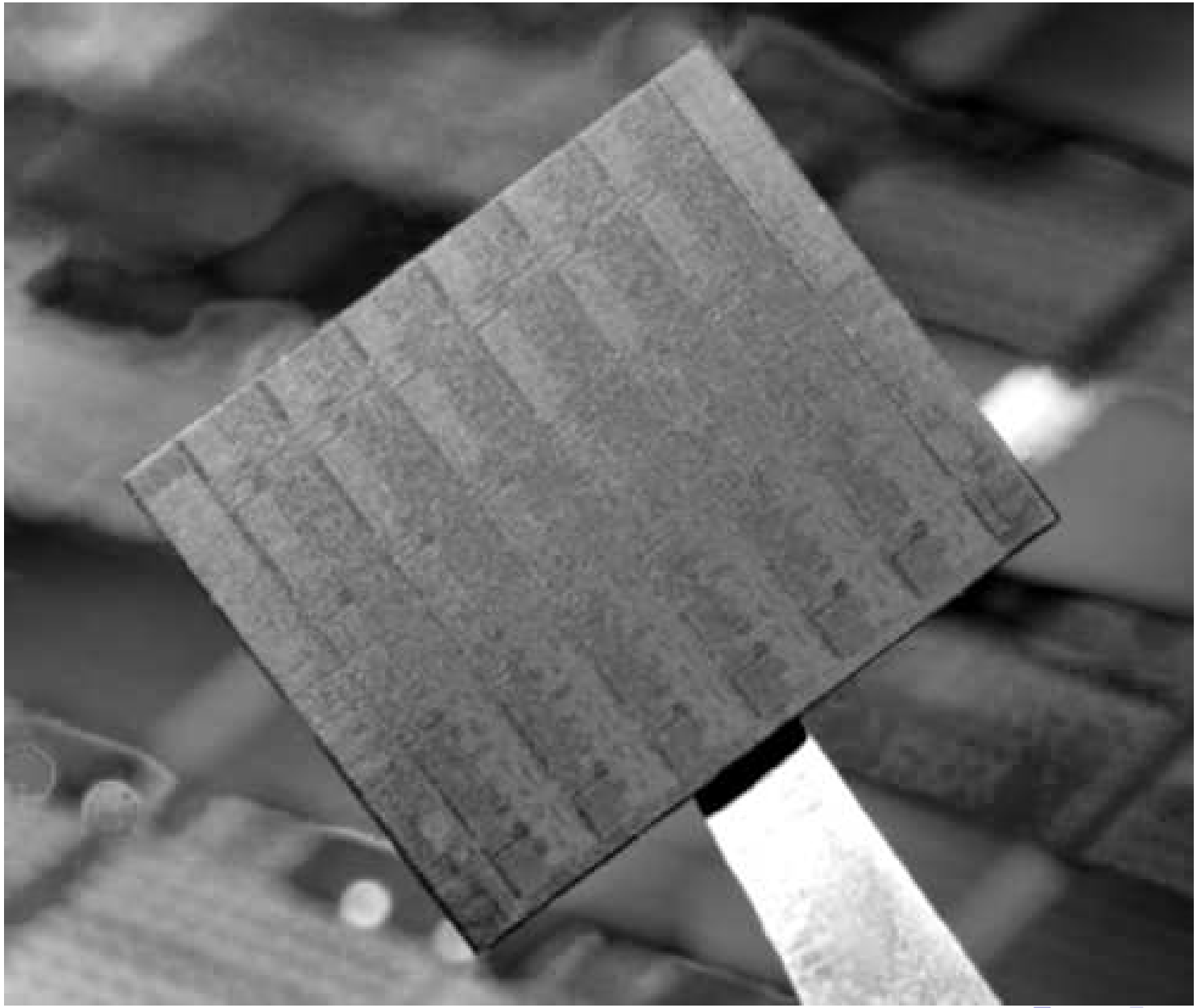
Moore's Law implies that as the scale of transistors shrinks, the number of gates that can be fitted onto a chip of a standard size, say of the order of $1cm^2$, will double every two years. Historically this has been used by processor manufactures to increase the complexity of individual processor cores.
A reduction in feature sizes potentially allows the speed of gates to rise, allowing a rise in clock speeds. This rise was pretty continuous until the last few years since when it has leveled off.

How parallelism is changing

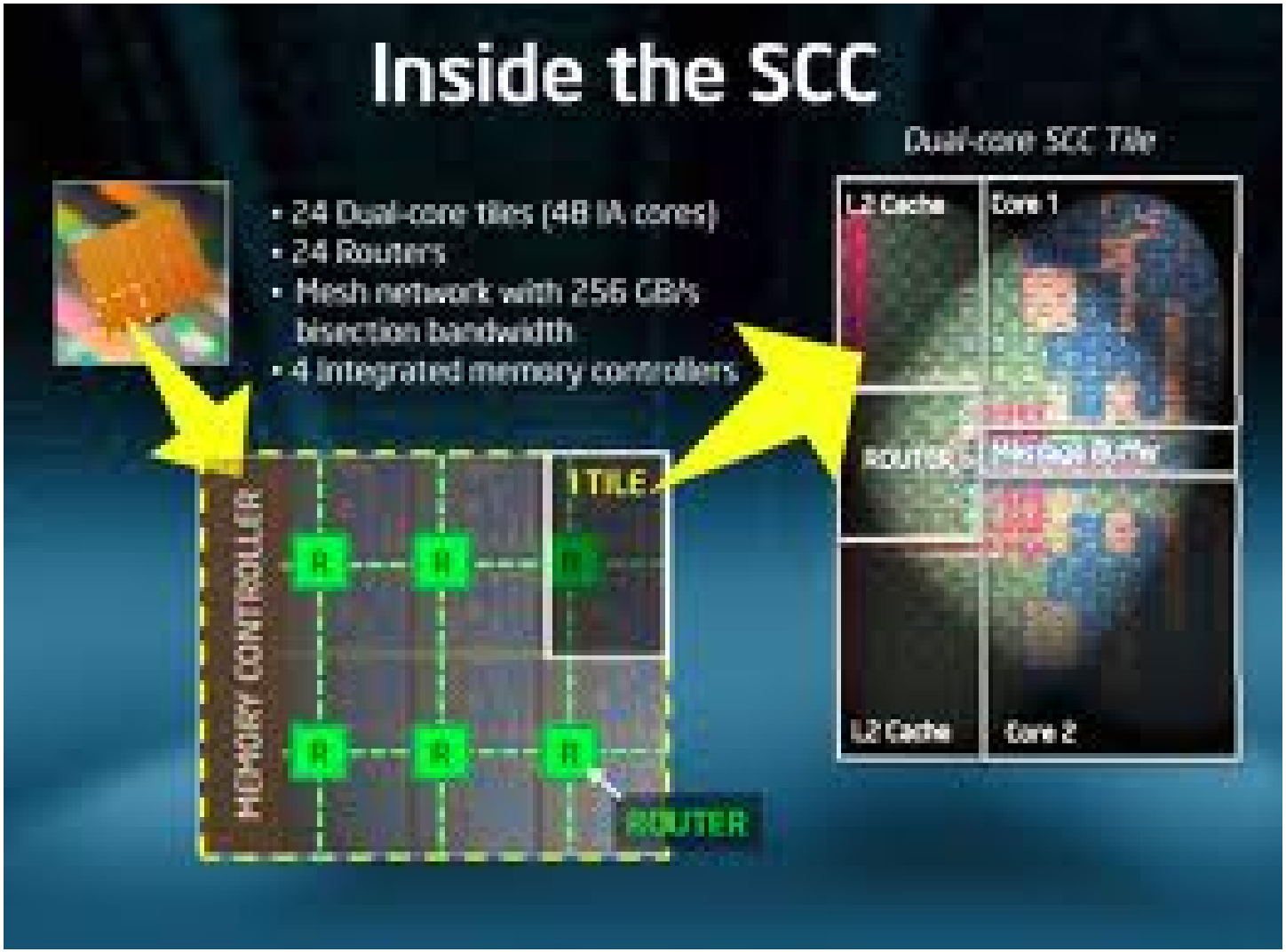
1. Higher clock speeds increase the heat dissipation per cm^2 due to capacitive losses, at around 3Ghz the heat losses are at the limit of what can be sustained with air cooling, even with heat pipes etc.
2. As clock speeds rise, clock skew accross the die becomes a significant factor which ultimately limits the ability to construct synchronous machines.

A result of these pressures has been that the mode of elaboration of chips has switched from complexifying individual cores, to the adding of multiple cores to each chip. We can now expect the number of cores to grow exponentially: perhaps doubling roughly every two years. This implies that in 10 years time a mass produced standard PC chip could contain around 256 or 512 cores.

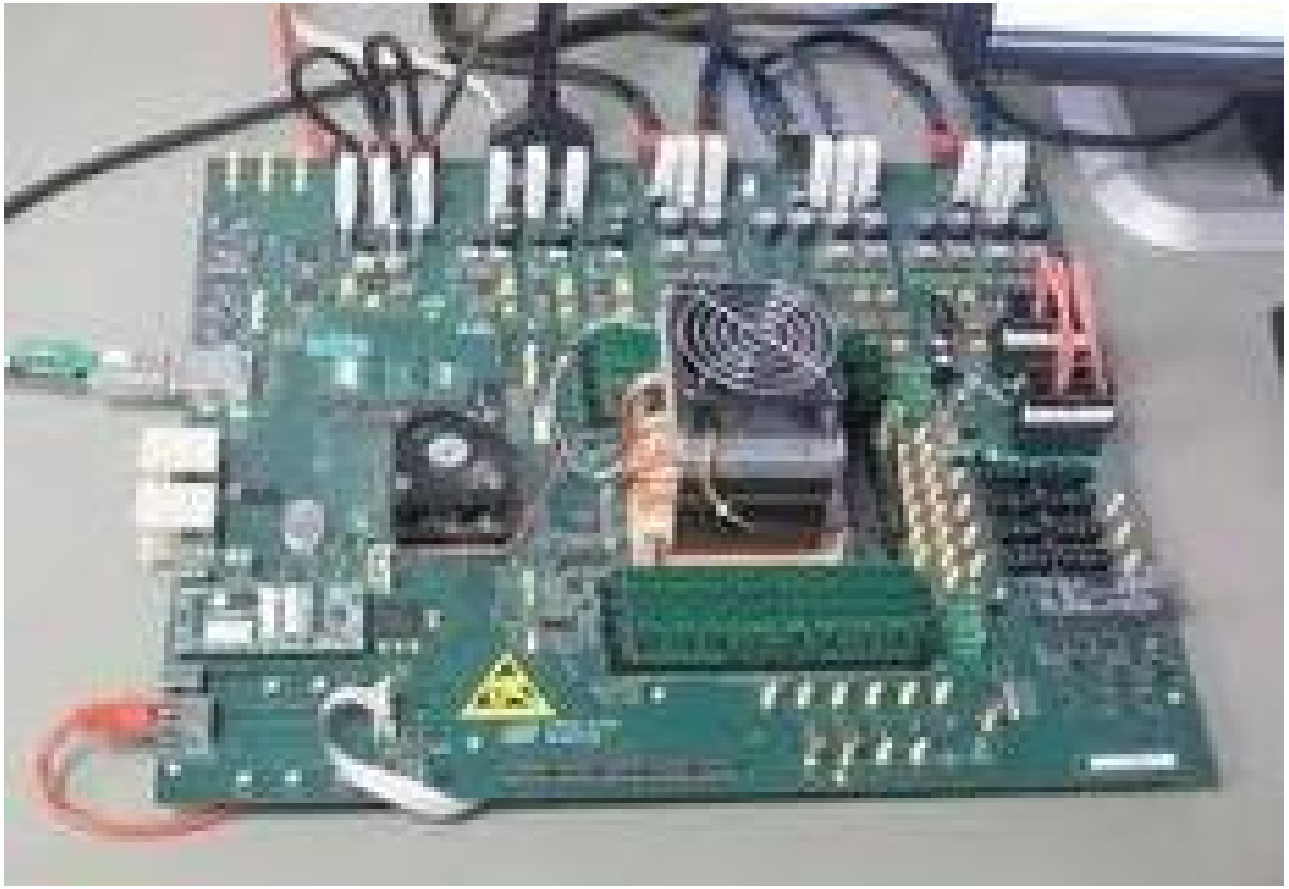
The SCC



Inside the SCC



The Development Board

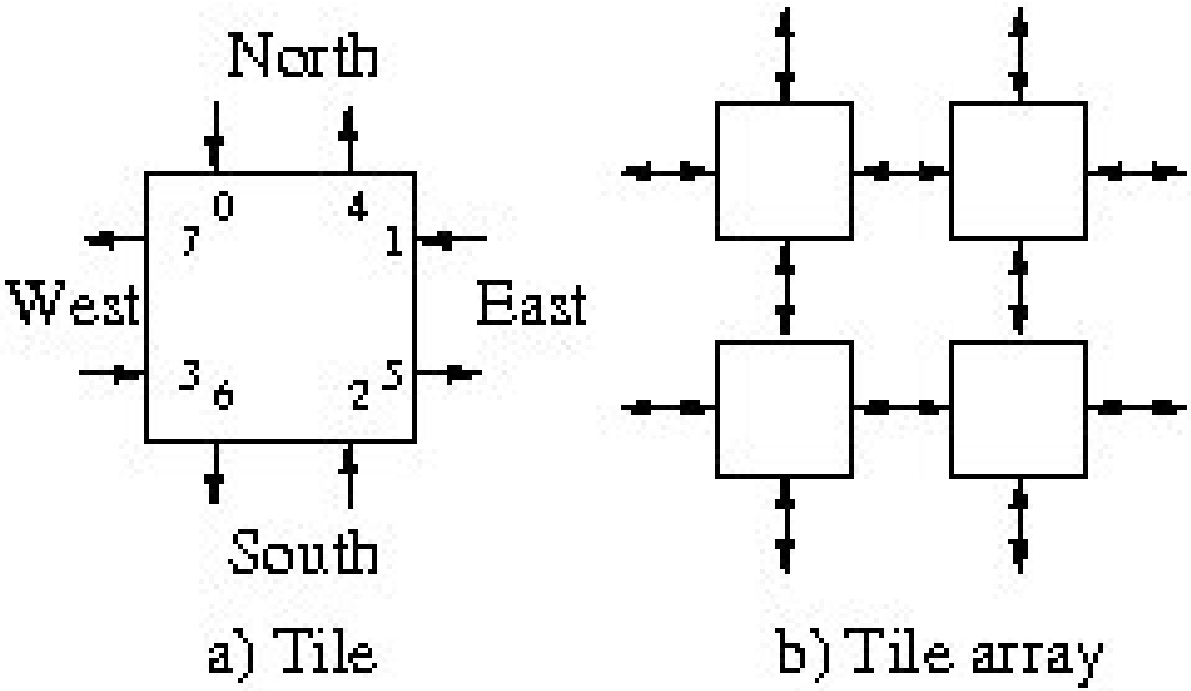


Need new types of languages

This growth in the number of cores and the problems of communicating between arbitrary processors is going to require a fundamental rethink in the way we design programming languages. In this talk I present Lino, a novel notation for programming arbitrarily large arrays of processors, based on abstractions over patterns of process adjacencies.

Lino Tiles and Tilings

Lino programs describe arrays of square tiles. Figure shows an atomic square tile and an array of tiles. A tile has one input stream and one output stream on each face, with inputs numbered `0..3` and outputs `4..7` in clockwise face order starting at the top\footnote{This is the convention for all face orderings.}. Faces are identified as *North*, *East*, *South* and *West*.



a) Tile b) Tile array

Syntax of Lino

$comm ::= dev \mid alias$	commands
$comms ::= comm[;comms]$	command seq
$prog ::= coms; main = exp$	a program is a sequence of commands ending with a nominated main expression
$def ::= id : faces <- path$	define tile id
$faces ::=$	
$((ty_0, ty_4), \ldots (ty_3, ty_7)$	I/O stream types
$ty ::= \ldots$	type
$path ::= \ldots$	file path
$alias ::= id = exp$	id aliases exp

A *command* is a tile *definition* or an *aliased* expression. A *definition* provides the tile name, the types of the input and output for each face, and a *path* to an executable body.

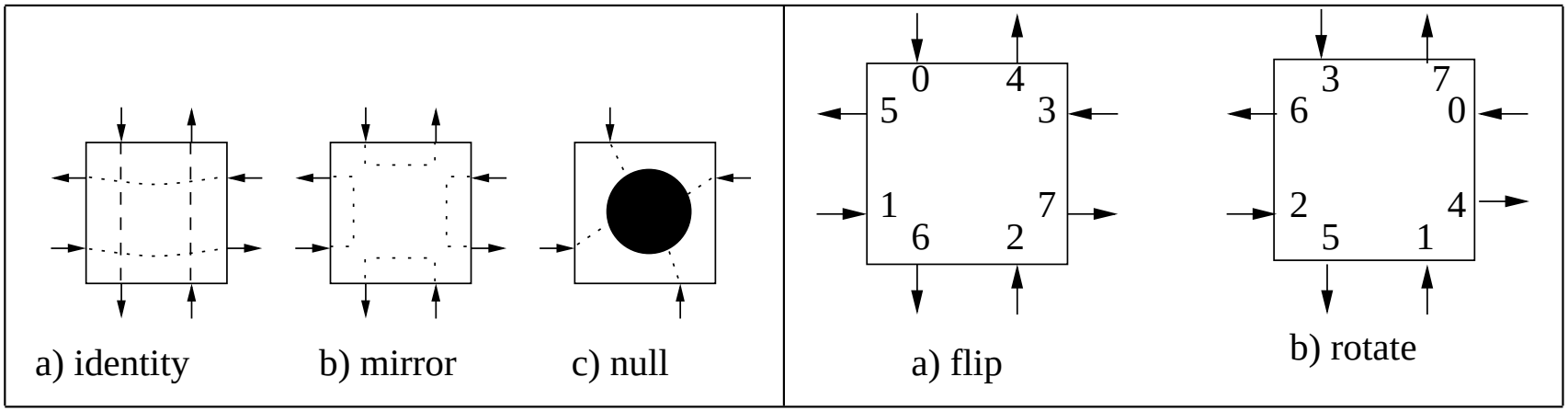
Syntax continued

block ::= **[** *redir*[;*redir*] **]**	shell block
redir::= *path dirio*[*dirio*]	redirected shell comma
dirio::= *inout direction*	
inout ::= < | >	standard redirections
direction::= `North` | `South` | `East` | `West`	direction names
exp::= ...	expressions
id	name of defined tile or aliased expression

More syntax

`I`	identity
`Mirror`	redirects face I/O
`0`	sink
`(` *exp* `)`	bracketing for priority
$exp_1 \mid exp_2$	process row
exp_1 `_` exp_2	process column
exp `*` *int*	horizontal replication
exp `^` *int*	vertical replication
`Flip` *exp*	reflection about vertical axis
`Rotate` *exp*	rotate 90 degrees clockwise

As in



Lino programs

A *program* is a sequence of *commands* ending with a nominated `main` expression. A *command* is a tile *definition* or an *aliased* expression. A *definition* provides the tile name, the types of the input and output for each face, and a *path* to an executable body.

Transform rules

	input	output
1	e*1	e
	e*N	e|(e*N-1)
2	e^1	e
	e^N	e^(e*N-1)
3	Flip I	I
	Flip Mirror	Mirror
	Flip 0	0
	Flip(e|f)	Flip f| Flip e
	Flip (e_f)	(Flip e)_(Flip f)

Transform rules continued

	input	output
4	Rotate I	I
	Rotate Mirror	Mirror
	Rotate 0	0
	Rotate (e|f)	(Rotate f) _ (Rotate e)
	Rotate (e_f)	(Rotate f)|(Rotate e)
5	Flip Flip e	e
6	Rotate Rotate Rotate Rotate e	e
7	(a_b)|(c_d)	(a|c)_(b|d)

What the rules mean

- The rules shown apply to expressions.
- Horizontal and vertical replication apply a fixed (and known) number of times (1 and 2).
- Flip and rotate preserve identity, mirror and null tiles (3 and 4).
- Flipping a row creates a row of flipped elements in reverse order; flipping a column creates a column of flipped elements (3).
- Rotating a row creates a column of rotated elements; rotating a column creates a row of rotated elements in reverse order (4)
- Two flips cancel (5).
- Four rotates cancel (6).
- Columns distribute over rows (7).

Status

A prototype implementation was completed last autumn.
Implementation proceeds in two stages. First, the main expression is fully expanded to column-major order. Then, the overall column of rows drives the generation of an equivalent shell script in which, for each tile position, approprite executable calls are made with stream redirection to linking FIFOs.
This first version runs on standard multi-core linux. It translates directly into shell script to generate the parallelism using & operations.
A new implementation is to be made targeted explicitly at the SCC.

An example script

```
lifecell:((int,int),(int,int),(int,int),(int,int)) <- ./
liferow = Mirror|(Flip  (lifecell *3) )|I|Mirror;
lifeblock = Flip (liferow ^ 3);
mirrorrow = Mirror * 6;
main = Rotate(mirrorrow _ lifeblock _ mirrorrow
```

The aim of the SICSA MultiCore Challenge is to compare several approaches to parallel computation in terms of achieved performance and ease of implementation. We plan to specify one or more representative applications to be implemented and assessed on state-of-the-art multi-core machines. We invite system developers to apply their systems on these benchmark applications. We invite software developers to put forward their applications as benchmarks for this challenge.
The first application proposed was a file concordance application.

Specification of the Concordance application.

Given: Text file containing English text in ASCII encoding. An integer N.
Find: For all sequences of words, up to length N, occurring in the input file, the number of occurrences of this sequence in the text, together with a list of start indices. Optionally, sequences with only 1 occurrence should be omitted.

Is it a good parallel problem?

I think this is a very hard programme to get good parallelism out of.
This is because a well designed serial programme to do concordance will spend a large part of its time reading in text or printing results.
This was not immediately apparent to the proposers, probably because they started out with a poorly written Haskell serial implementation.
When looking at any problem the first thing to do is get an estimate of the complexity order of the problem.
My intuition was that this was roughly $\mathbf{O}(N)$.

Quick Hack

Prior to doing any parallelisation it is advisable to initially set up a good sequential version. I was initially doubtfull that this challenge would provide an effective basis for parallelisation because it seemed such a simple problem. Intutively it seems like a problem that is likely to be of either linear or at worst log linear complexity, and for such problems, especially ones involving text files, the time taken to read in the file and print out the results can easily come to dominate the total time taken. If a problem is disk bound, then there is little advantage in expending effort to run it on multiple cores.

However that was only an hypothesis and needed to be verified by experiment. In line with our school motto of programming to an interface not an implementation, the interface above rather than the Haskell implementation was chosen as the starting point. In order to get a bog standard implementation, C was chosen as the implementation language.

Serial results

The first thing to note is the C is much faster than the initial Haskell. The difference in speed is far greater than could be accounted for in terms of the relative efficiencies of the compilers. Instead it indicates that the Haskell is a poor algorithm.

Initial runs on windows

version	input file size	time
haskell	3kb	0.82 sec
c	3kb	0.24 sec
haskell	4.9mb	timed out after 3 hours
c	4.9mb	3.67 sec

Algorithm Structure

The algorithm used had the following basic structure

1. Read the inut file to a buffer.
2. Tokenize it to a sequence of integers corresponding to words.
3. Make a single pass through the tokenized data building a hashed index.
4. Make a final pass throught the index printing out the results.

It is clear that this algorithm is basically or order N as it has 3 sequential passes. andd that steps 1 and 4 are likely to take a significant fraction of the time.
How can it be parallelised accross cores?

What you can not do

You can not simply split the file into two halves, produce a concordance for each half and merge them. The aim is to produce a list of words and positions for all repated words. If you split the file in two halves you are not guaranteed to find repetitions unless you do something smart.
So how to proceed?
Recall we tokenize the file mapping it to integers.
How about two processes one of which deals with all the odd words and one with all the even words?

Two versions

I tried two versions

- Use pthreads, read file in and tokenize once, then get two processes to go through and build two disjoint indices and print them to two files. Then use cat to join the files.
- Use the shell & operator to run two copies of the original C lightly modified to process either even or odd words to two files and cat them.

How did these perform:
Tests were done on Linux using the same processor as the previous example, using the 4.9meg World English Bible as data. This time the C was optimised with -O3

example	runtime
serial version in C	2.12 secs
dual thread version using Pthreads	2.45 secs
dual process version using bash	1.93 secs

Lessons

- The old bash shell is actually better than pthreads
  - this is good news for Lino since it compiles to bash shell commands
- Overall speedup minor because task i/o limitedhat

What next

- On the SCC
  - Try to run the concordance bash style on say 32 cores
  - Port Lino to the SCC
- On the SICSA front
  - Encourage you all to try your hand at it
  - Report results in december
  - Propose better examples to SICSA
    - Mandelbrot
    - Convolution
    - Disparity matcher