Lino: a tiling language for arrays of processors

Paul Cockshott
Department of Computer Science
University of Glasgow
wpc@dcs.gla.ac.uk

Greg Michaelson
School of Mathametical and Computer Sciences
Heriot-Watt University
G.Michaelson@hw.ac.uk

Lino is language for tiling large arrays of processor, in particular for multi-core. Lino is oriented to the coordination and communication aspects of multi-processing, and is otherwise implementation neutral, thus naturally facilitating the composition of large systems from heterogeneous software components. The need for Lino is motivated, and Lino's design and implementation are surveyed.

Lino, multi-core, tiling, coordination language

1. INTRODUCTION

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. The word length has grown from 4 bits in the first chips back in the 1970s to 128 bits on Intel machines circa 2000. In an orthogonal direction, the pipeline depths have increased, as has the logic to exploit instruction level parallelism: dual isssue, out of order issue, etc.

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. The levelling off has been due to :

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.

As if the growth in the number of cores were not enough to worry about, one also has to think of what the implications of this are for memory access. Although it has proven possible to shrink the feature sizes on chip, providing an increased number of pins comming off the chip has proven much harder. The number of pins on chips has grown, but at slower exponential rate than the number of gates on chip. If this trend continues, then it will be increasingly difficult to provide all of the cores on the chip with an adequate bandwidth to a single unified main memory. Caches can help up to a degree with this, but they can only disguise and partially delay the onset of a bandwidth bottle-neck in communication with a unified main store. We are likely to see an increasing reliance on each of the cores having its own private high speed memory on chip. We can already see this with the Cell in which each vector core has its own completely private memory. The Intel Larabee attempts to make these.

Most recently, Intel has announced an experimental CPU with 48 cores, as shown in Figure 1. This is intended to support what they term "Single-chip Cloud Computing".

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 paper, we present Lino, a novel notation for programming arbitrarily large arrays of processors, based on abstractions over patterns of process adjacencies. In the following sections we explore the limitations of current approaches, present the Lino designs and explore a first Lino implementation.

2. BACKGROUND

Multi-processing requirea programmings mechanisms for creating processes, communicating between them and coordinating their behaviours.

© *The Authors. Published by the British Informatics Society Ltd.*

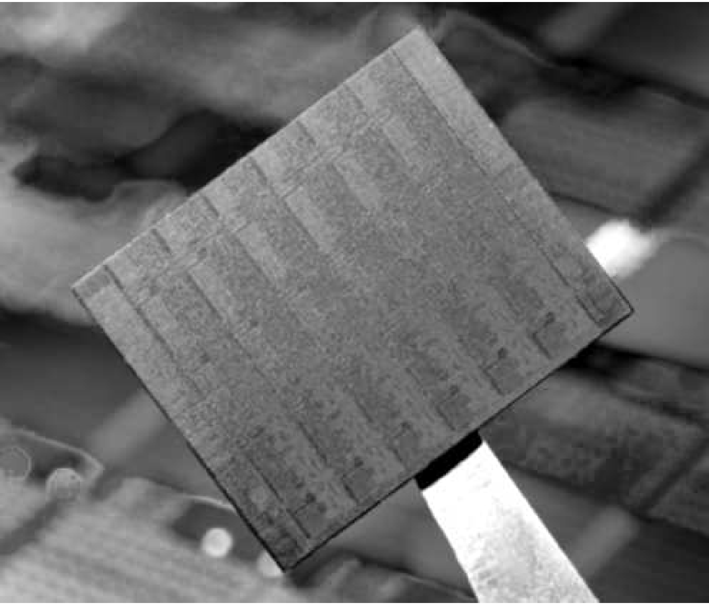

Figure 1: *Intel Rock Creek CPU.*

We may discern three main approaches to such programming. Fully automated parallelisation aims to extract useful parallelism implicit in programs. Here the compiler and run-time system are responsible for all apsects of process creation, communication and coordination. Despite many years research, most success has still been enjoyed with loop/array parallelisation in Fortran(KL+94). Even with very sophisticated program analysis, in general parallism is restricted to regular patterns of processing across regular data structures.

A second approach is to develop new languages with integrated parallel constructs. There have been numerous proposals for parallel programming languages. However, because of the enormous investment required to deploy and maintain a new language in the teeth of existing code base investments, few make it beyond the PhD without significant industrial support. Probably the most succesful contemporary parallel language is Erlang(AVWW96). The highly promising Occam(Inm87), which died for reasons of economy rather than technology, remains very influential.

Typically, such languages offer new constructs for expressing patterns of processing and communication. Thus both Occam and Erlang provide explicit processes with explicit channel communication. In addition, Occam provided the potentially language independent notion of wiring where processes are explicitly associated with processors.

The contemporary Hume(Hammond03) has concurrent boxes as the locus of processing with explicit wire interconnections. Hume is novel in explicitly separating a rich control language for describing individual boxes from a simple coordination language for describing their interconnections. Hume is also novel in offering implicit type driven inter-process communication: data of known type is automatically serialised and unserialised without programmer intervention. Like Occam, however, Hume is a flat language and lacks abstractions over processes and communication.

The most common approach to programming multi-processor systems is through specialised libraries in extant languages. Such libraries may be language and platform independent, as with MPI(Gro00) and PVM(PVM93) for distributed memory systems based on high speed interconnects, Pthreads(LB98) and OpenMP(CJP07) for shared memory systems now typically based on multi-core assemblies. There are also language specific, platform independent libraries, like the very widely used Java threads(Hyd99), and platform specific, language independent libraries, like Intel threads (Rei07). Such libraries tend to be relatively low level. They also tend to require deep understanding of an often opaque multi-processor model, and its interactions with the host programming language. As with parallel languages, in the absence of appropriate abstraction mechanisms, it is not clear how such libraries use can scale effectively to open-endedly large systems.

An approach increasingly adopted by large companies, to manage processing on very large farms of processors, is to provide highly scalable but architecturally restricted frameworks for parallelism. These often draw on ideas from the algorithmic skeleton community(Cole02) to provide standard patterns of multi-process computation. Thus, Google uses their proprietary map-reduce(Dean08) approach, where Yahoo has adopted the closely comparable open-source Hadoop(Apache08) from Apache. In turn, Microsoft deploys the somewhat more general Dryad framework(Isard07).

3. LINO DESIGN

We think that the increasing heterogeneity of contemporary systems, both in software and hardware components, requires a high degree of language and platform independence. In turn, the growing complexity of contemporary systems requires a very high degree of automation; programmers should focus on defining computations and their patterns at an abstract level and the implementation should realise the underlying glue. Next, the sheer scale of contemporary systems, as seen in industrial processor farms of tens of thousands of CPUs, requires some restriction in the generality of any viable approach to constructing them. Finally, we observe a high degree of regularity of pattern of computational elements in contemporary systems, where complex computations at a low level are massively replicated in simple formations at a high level.

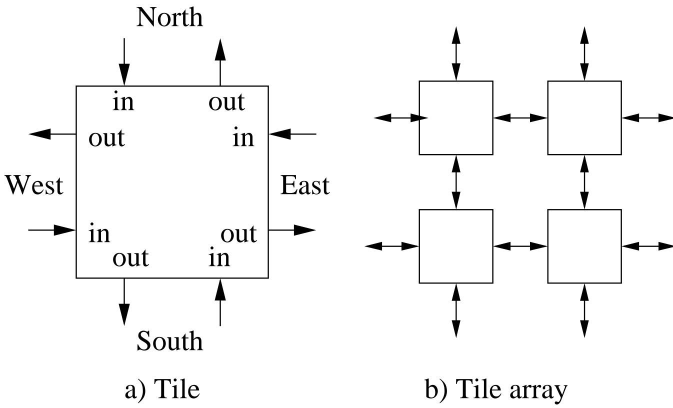

Figure 2: *Tile and tile array*

Thus, our new language Lino [1] is aimed at:

- tiling large conceptual two-dimensional arrays of procesing elements with patterns of computations;
- facilitating hierachies of pattarns of such arrays;
- language and platform independent implementation;
- minimising "glueware" between procesing elements.

The implications of these design decisions are explored in the following sections.

4. TILES AND TILINGS

Lino programs describe arrays of square tiles. Figure 2 shows an atomic square tile and an array of tiles. A tile has one input stream and one output stream on each face. Faces are identified as $North$, $East$, $South$ and $West$.

Figure 3 shows the syntax of Lino.

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

Atomic tiles, shown in Figure 4 include identity (`I`), which sends all inputs to the outputs on the opposite faces, `Mirror`, which sends the input to the output on each face, and null (∅), which has no outputs and absorbs inputs.

Tile constructs may be placed next to each other with the operators `|` to form rows and `_` to form columns. Tile constructs may be replicated horizontally (`*`), to forms rows, and vertically (`^`), to form columns, a fixed integer number of times.

[1] c.f. linoleum tiles.

$comm$ ::= def | $alias$	commands
$comms$ ::= $comm$ [; $comms$]	command sequence
$prog$ ::= $comms$; `main` = exp	program
def ::= id : $faces$ <- $path$	define tile id
$faces$ ::=	
((ty_0,ty_4),...(ty_3,ty_7))	I/O stream types
ty ::= ...	type
$path$ ::= ...	file path
$alias$ ::= id = exp	id aliases exp
$block$ ::= **[** $redir$[;$redir$]**]**	shell block
$redir$ ::= $path$ $dirio$ [$dirio$]	redirected shell command
$dirio$ ::= $inout$ $direction$	
$inout$ ::= > | <	standard i/o redirection
$direction$ ::= `North`|`East`|`South`|`West`	face labels
exp ::= ...	expressions
id	name of defined tile
	or aliased exp
`I`	identity
`Mirror`	reflects face I/O
∅	sink
(exp)	brackets
exp_1 | exp_2	row
exp_1 _ exp_2	column
exp * int	horizontal replication
exp ^ int	vertical replication
`Flip` exp	horizontal mirror image
`Rotate` exp	rotate 90 degrees clockwise

Figure 3: *Lino syntax*

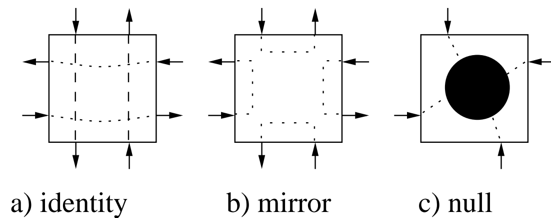

Figure 4: *Atomic tiles*

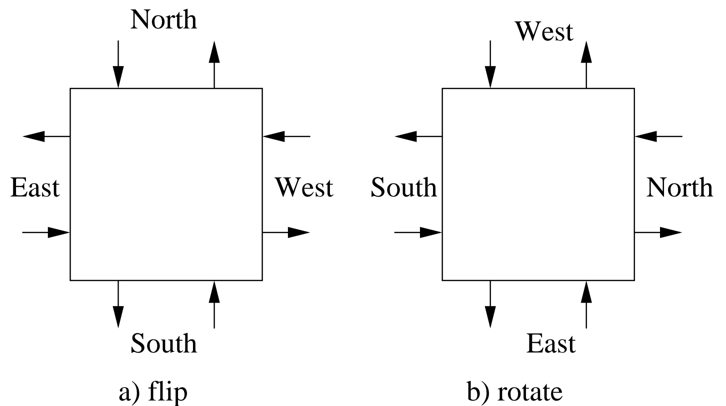

Figure 5: *Flip and rotate*

1.	`e*1`	$\Rightarrow$	e
	`e*N`	$\Rightarrow$	e | (e*N-1)
2.	`e^1`	$\Rightarrow$	e
	`e^N`	$\Rightarrow$	e_(e^N-1)
3.	`Flip I`	$\Rightarrow$	`I`
	`Flip Mirror`	$\Rightarrow$	`Mirror`
	`Flip` $\emptyset$	$\Rightarrow$	$\emptyset$
	`Flip` (e | f)	$\Rightarrow$	Flip f | Flip e
	`Flip` (e_f)	$\Rightarrow$	Flip f_Flip e
4.	`Rotate I`	$\Rightarrow$	`I`
	`Rotate Mirror`	$\Rightarrow$	`Mirror`
	`Rotate` $\emptyset$	$\Rightarrow$	$\emptyset$
	`Rotate` (e | f)	$\Rightarrow$	Rotate f_Rotate e
	`Rotate` (e_f)	$\Rightarrow$	Rotate f | Rotate e
5.	`Flip Flip` e	$\Rightarrow$	e
6.	`Rotate Rotate` `Rotate Rotate` e	$\Rightarrow$	e
7.	(a_b) | (c_d)	$\Rightarrow$	(a | c)_(b | d)

Figure 6: *Transformation rules*

Finally tile constructs may be `flip`ed through 180 degress horizontally, or `rotate`d clockwise through 90 degrees, with the corresponding reordering of face/stream correspondences. See Figure 5.

5. TRANSFORMATION RULES

The rules shown in Figure 6 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).

6. IMPLEMENTATION

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.

To expand a main expression:

1. replace every alias identifier with the corresponding expresssion;
2. expand up all replication;
3. push all flips and rotates inwards;
4. cancel flips and rotates;
5. distribute columns over rows.

The effect of this is to reduce the tree to normal form comprised of horizontal and vertical join operations on tiles which may themselves contain flip and rotate operations.

A depth first traversal of the tree is then performed to build up a two dimensional array of these atomic tile expressions.

An atomic tile translates into a [1,1] tile array. x|y will result in an [n,m] tile array when the translation of x is an [n,p] tile array and that of y an [n,q] tile array with m=p+q and generates a tiling error otherwise.

Similarly x_y gives an [n,m] tile array where the translation of x is an [p,m] tile array and that of y an [q,m] tile array assuming n=p+q and again generates a tiling error otherwise.

Given a tiling array, it can be translated into a shellscript in the following stages.

1. Generate 4 output fifos for each tile thus:

   ```
   mkfifo fifos/North0_0
   mkfifo fifos/East0_0
   mkfifo fifos/South0_0
   mkfifo fifos/West0_0
   mkfifo fifos/North0_1
   mkfifo fifos/East0_1
   mkfifo fifos/South0_1
   mkfifo fifos/West0_1
   ```

   etc which generates fifos for row 0, tiles 0 and 1. These Linux fifos are to all appearances just files in the file system. They can be opened by two process one as a reader and the other as a writer, using normal file open calls and the operating system then ensures that they operate as blocking, buffered channels.

2. Then for each tile allocate a 4 element array of output fifonames. The elements of this array initially correspond to "North","East","South","West".

3. Next visit each tile and perform any flips and rotates that are required on the array of output fifos associated with that tile. A rotate is done by a cyclic shift of the array, and a flip by exchanging the East and West fifo names.

4. Next create an empty 4 element array of input fifo names for each tile. Initialise these by following a path out from the current tile, normal to the edges until you encounter a tile capable of generating output. Thus you must pass over identity cells.

   A mirror cell counts as a potential source of data as it is actually implemented as 1 or more processes

that copy output to input. It might appear that one could simply reverse ones path on reaching a mirror cell, but that would result in a source process both writing to and reading from a fifo, which would cause it to hang when trying to open the fifo.

5. Apply the flips and reverses to the 4 element array of input fifo names just as they were applied to the output fifo names.

6. As a last stage traverse the tiling and for every non routing tile output a shell command to run the process with appropriate i/o redirection and command line parameters specifying the positions in the array at which the tile is running. The successive shell commands are separated by &s to spawn parallel processes.

7. EXAMPLE: GAME OF LIFE ON 8 CORES

Consider an implementation of the Game of Life, written in C. Suppose now we wish to run it on an eight-core platform such that there are, conceptually, four rows of two life tiles. We surround the central eight life tiles with mirror tiles to provide a boundary condition:

```
lifecell:((int,int),(int,int),(int,int),(int,int))
 <- ./lifeprog;
liferow = Mirror | lifecell | lifecell | Mirror;
mirrorrow = Mirror * 4;
main =  mirrorrow _ (liferow^4) _ mirrorrow;
```

A `lifecell` has integer input and output streams on all faces and is implemented by the exceutable program `lifeprog`. A `liferow` is two `lifecell`s bounded by `Mirror`s. A `mirrorrow` is four `Mirrors`. The whole tileing (`main`) is four `liferow`s between two `mirrorrow`s.

In this case, the expansion to normal form is straightforward. First of all, macro-substitution occurs to replace all occurrences of `mirrorrow` and `liferow`:

```
main = Mirror * 4 _
       (Mirror | lifecell | lifecell | Mirror) ^ 4 _
       Mirror * 4
```

And next, the replications are expanded out, to reveal the final tiling:

```
main = Mirror | Mirror   | Mirror   | Mirror _
       Mirror | lifecell | lifecell | Mirror _
       Mirror | lifecell | lifecell | Mirror _
       Mirror | lifecell | lifecell | Mirror _
       Mirror | lifecell | lifecell | Mirror _
       Mirror | Mirror   | Mirror   | Mirror
```

The second stage, compilation, generates FIFOs as above, followed by a shell program. The shell script generated by the compiler for this example is shown in Figure 7.

Note that the corner mirrors are ommitted.

```
cat   < fifos/North1_1  > fifos/South0_1  &
cat   < fifos/North1_2  > fifos/South0_2  &
cat   > fifos/East1_0  < fifos/West1_1  &
./lifeprog   8< fifos/East1_0  3> fifos/East1_1
             1> fifos/North1_1  6< fifos/North2_1
             0< fifos/South0_1  5> fifos/South1_1
             7> fifos/West1_1  4< fifos/West1_2  1 1&
./lifeprog   8< fifos/East1_1  3> fifos/East1_2
             1> fifos/North1_2  6< fifos/North2_2
             0< fifos/South0_2  5> fifos/South1_2
             7> fifos/West1_2  4< fifos/West1_3  1 2&
cat   < fifos/East1_2  > fifos/West1_3  &
cat   > fifos/East2_0  < fifos/West2_1  &
./lifeprog   8< fifos/East2_0  3> fifos/East2_1
             1> fifos/North2_1  6< fifos/North3_1
             0< fifos/South1_1  5> fifos/South2_1
             7> fifos/West2_1  4< fifos/West2_2  2 1&
./lifeprog   8< fifos/East2_1  3> fifos/East2_2
             1> fifos/North2_2  6< fifos/North3_2
             0< fifos/South1_2  5> fifos/South2_2
             7> fifos/West2_2  4< fifos/West2_3  2 2&
cat   < fifos/East2_2  > fifos/West2_3  &
cat   > fifos/East3_0  < fifos/West3_1  &
./lifeprog   8< fifos/East3_0  3> fifos/East3_1
             1> fifos/North3_1  6< fifos/North4_1
             0< fifos/South2_1  5> fifos/South3_1
             7> fifos/West3_1  4< fifos/West3_2  3 1&
./lifeprog   8< fifos/East3_1  3> fifos/East3_2
             1> fifos/North3_2  6< fifos/North4_2
             0< fifos/South2_2  5> fifos/South3_2
             7> fifos/West3_2  4< fifos/West3_3  3 2&
cat   < fifos/East3_2  > fifos/West3_3  &
cat   > fifos/East4_0  < fifos/West4_1  &
./lifeprog   8< fifos/East4_0  3> fifos/East4_1
             1> fifos/North4_1  6< fifos/North5_1
             0< fifos/South3_1  5> fifos/South4_1
             7> fifos/West4_1  4< fifos/West4_2  4 1&
./lifeprog   8< fifos/East4_1  3> fifos/East4_2
             1> fifos/North4_2  6< fifos/North5_2
             0< fifos/South3_2  5> fifos/South4_2
             7> fifos/West4_2  4< fifos/West4_3  4 2&
cat   < fifos/East4_2  > fifos/West4_3  &
cat   > fifos/North5_1  < fifos/South4_1  &
cat   > fifos/North5_2  < fifos/South4_2  &
```

Figure 7: *Shell script generated by the lino compiler from the Game of Life example*

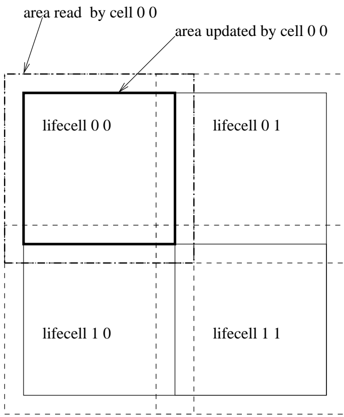

Figure 8: *Each process has an area it updates and a larger area it reads from, which overlaps those written by other processes.*

The programme `lifeprog` maintains a 2D array of cells which keep the current states of a square region of the plain of automata that are being emulated. It could be written in a variety of languages, but in what follows we give an example in C.

Its basic control structure is a loop which performs communication and then updates the states of the automata:

```
for(count=0;count<ITER;count++)
  {    /* communications */
       writestates();
       readstates();
       exchangecorners();
       /* updateing */
       updatestates();
  }
```

As Figure 8 shows, each process has an area of the array which it updates and a larger area that it reads from. The array of states is defined as:

```
char state[EDGE+2][EDGE+2];
```

to allow for this. The subarray `state[1..EDGE][1..EDGE]` is actually updated by the program, the outer rows and outer columns are obtained from it's neighbours. A process talks over the garden fence to it's neighbours is via fifos. To prevent blocking, each process must write before it reads. A secondary sequence of io operations, `exchangecorners()`, is needed to access the diagonal corner cells, as these come from processes with which the current process is not in direct contact. The actual reading and writing is done using standard Posix system calls as shown in Figure 9. Thus no special runtime library or harness is needed to run C code within a lino tiling.

```
writestates()
{
    char westbuf[EDGE];char eastbuf[EDGE];int i;
    /* write out rows */
    write( NORTHOUT,&(state[1][1]),EDGE);
    write( SOUTHOUT,&(state[EDGE][1]),EDGE);
    /* copy cols to buffers */
    for(i=0;i<EDGE;i++){
       westbuf[i]=state[i+1][1];
       eastbuf[i]=state[i+1][EDGE];
    }
    write( EASTOUT,eastbuf,EDGE);
    write( WESTOUT,westbuf,EDGE);
}
```

Figure 9: *Input and output from fifos is done using standard system calls, with the channels NORTHOUT etc, being defined by the include file* `lino.h`.

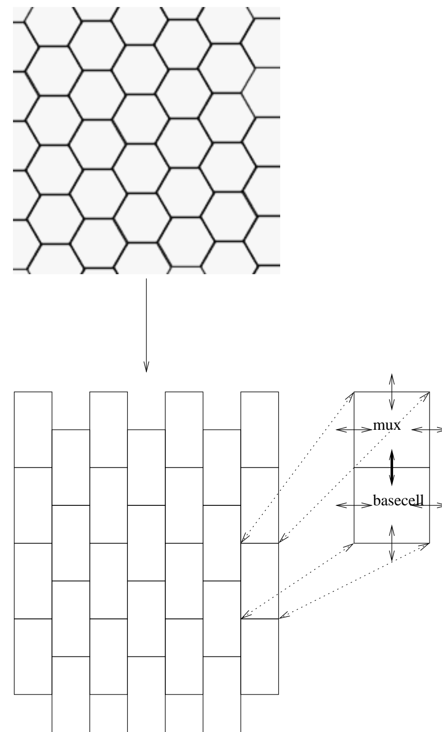

Figure 10: *Mapping a hexagonal tiling to a rectilinear one. In both tilings each tile has 6 neighbours.*

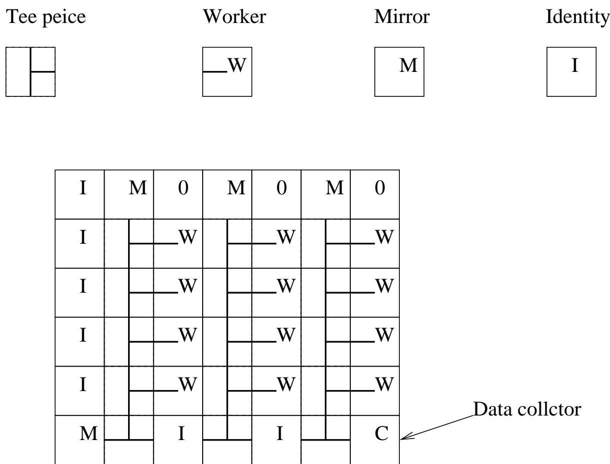

Figure 11: A process farm tiling, with worker cells W and Tee pieces that connect these to the data collector C.

8. OTHER TESSELATIONS

It may be necessary in some cases to have a non-square tiling. For example simulating latice gasses can require a hexagonal tiling. A hexagonal tiling can be mapped to a rectilinear tiling with an equivalent adjacency graph as shown in Figure 10. The basic components are rectangles with 6 connecting fifos which are then laid out offset by one cell position. The basic rectangles can be expressed in lino as `hex= mux _ basecell;`, where `mux` is a process that multiplexes its west, north and east inputs to its south output and vice versa. The `basecell` can then have communications with six neighbours as required for the hexagonal adjacency graph.

One can also construct processor farm tesselations with a T piece

T= ((char,char),(char,char),(char, char),(,))teemux

The teemux program runs the following algorithm:

1. Read a character c (either a 0, or a 1) from the south.
2. Write the character c to the north and read the reply character r_N from the north.
3. If $r_N = 0$ it means no results are available so goto step 5.
4. If $r_N =$1 read a count byte n, followed by n bytes of data from the north.
5. Write the sequence (1,n, bytes^n) to the south and goto 1
6. Write the character c to the east. Read reply from the east r_E.
7. If $r_E = 0$ it means no results are available write 0 to the south and terminate..
8. If $r_E =$1 read a count byte n, followed by n bytes of data from the east.
9. Write the sequence (1,n, bytes^n) to the south and goto 1.

Cores	Tiles	Time (s)	8 tiles/2 tiles
2	2	27.14	
2	8	106.80	3.94
8	2	14.68	
8	8	14.88	1.01

Figure 12: Times obtained for running different numbers of lifecells in parallel on a 2 and an 8 core machine.

This attempts to suck data from the north until there is no more available, and then such it from the east until there is no more available. By composing these with worker cells to their east, it is possible to build up a network of tiles that will route the results produced by the worker cells in a determinate order to a collector cell. An illustration of such a layout is given in Figure **??**.

9. EVALUATION

The life program was timed, with each tile processing a 512∗512 grid, on a two-core and an eight-core CPU. Figure 12 shows the times.

We observe that, on the two-core machine, eight life tiles take roughly four s times as long as two life tiles, and that, on the eight-core machine, both two and eight life tiles take roughly the same time. Of course this is a highly uniform example with highly regular processing on each tile. Nonetheless, these preliminary times suggest that our simple implementation scheme is robust.

10. CONCLUSION

11. REFERENCES

[Apache08]Apache. *Hadoop Map/Reduce Tutorial*. Apache Software Foundation, 2008.

[AVWW96]J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in* ERLANG. Prentice Hall, $2^{\rm nd}$ edition, 1996.

[CJP07]B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[Cole02]M. Cole. Bringing Skeletons out of the Closet. *Parallel Computing*, 30(3):389–406, 2004.

[Dean08]J. Dean and S. Ghemawat. MapReduce:SimplifiedDataProcessingonLargeClusters. *Communications of the ACM: 50th anniversary issue: 1958 - 2008, Special Issue: Breakthrough research:*

a preview of things to come, 51(1):107–113, January 2008.

[Gro00]W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT, 2000.

[Hammond03]K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng., Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.

[Hyd99]P. Hyde. *Java Thread Programming*. Sams, August 1999. ISBN 0-672-31585-8.

[Inm87]Inmos. *Occam2 Reference Manual*. Prentice-Hall, 1987.

[Isard07]M. Isard ,M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of European Conference on Computer Systems (EuroSys)*, pages 59–72, March 2007.

[KL+94]C. H. Koebel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr, and M. E Zosel. *The High Performance Fortran Handbook*. MIT, 1994.

[LB98]B. Lewis and D.J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[PVM93]*Parallel Virtual Machine Reference Manual*. University of Tennessee, Aug 1993.

[Rei07]J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'REILLY, 2007.