ICOT Technical Report: TM - 0031

TM-0031

Lecture Notes on
The Bagel: a Systolic Concurrent Prolog Machine

by

Ehud Shapiro
The Weizmann Institute of Science
Rehovot, ISRAEL

November, 1983

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

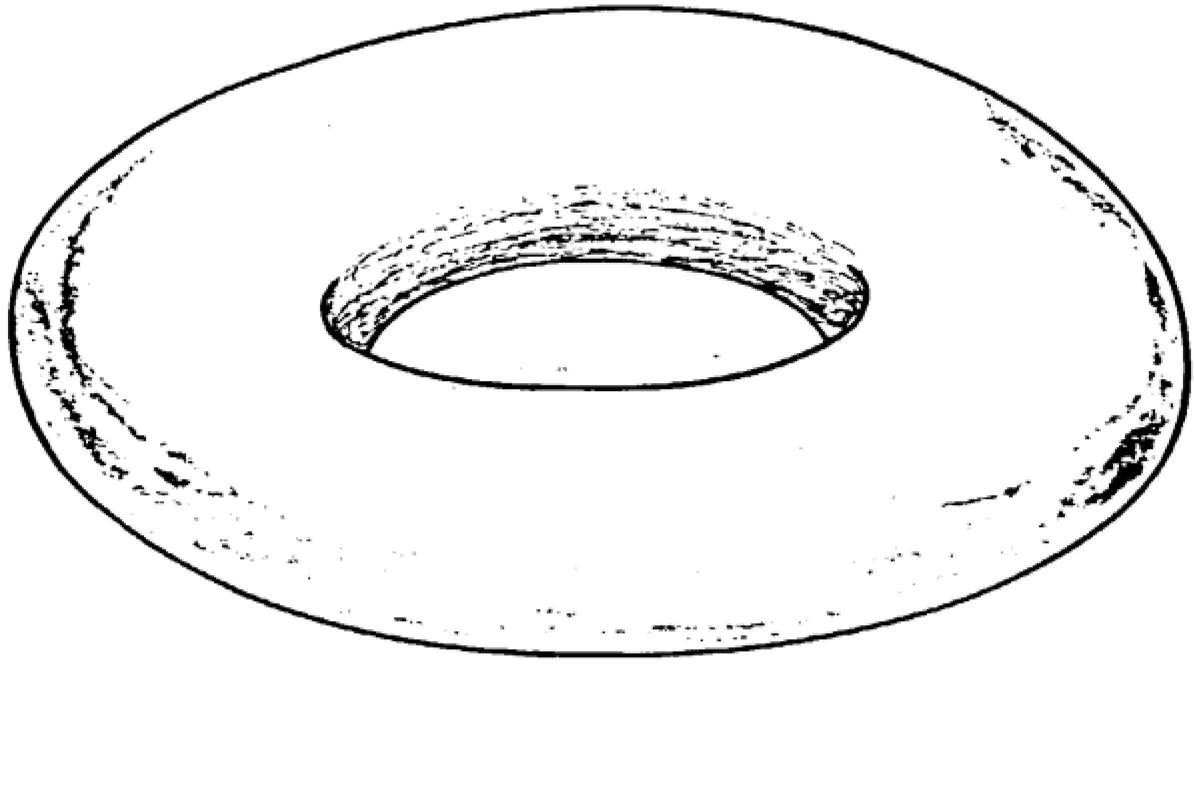
(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

The Bagel: A Systolic Concurrent Prolog Machine

Ehud Shapiro
The Weizmann Institute of Science
Rehovot, ISRAEL

November, 1983



The Bagel: A Systolic Concurrent Prolog Machine
(Lecture Notes)

Ehud Shapiro
The Weizmann Institute of Science
Rehovot, ISRAEL

November, 1983

Abstract

It is argued that explicit mapping of processes to processors is essential to effectively program a general-purpose parallel computer, and, as a consequence, that the kernel language of such a computer should include a process-to-processor mapping notation.

The Bagel is a parallel architecture that combines the concepts of dataflow, graph-reduction, and systolic arrays. The Bagel's kernel language is Concurrent Prolog, augmented with Turtle programs as a mapping notation.

Concurrent Prolog, combined with Turtle programs, can easily implement systolic systems on the Bagel. Several systolic process structures are explored via programming examples, including linear pipes (sieve of erasthotenes, merge sort, natural-language interface to a database), rectangular arrays (rectangular matrix multiplication, band-matrix multiplication, dynamic programming, array relaxation), static and dynamic H-trees (divide-and-conquer, distributed database), and chaotic structures (a herd of Turtles).

All programs shown have been debugged using the Turtle graphics Bagel simulator, which is implemented in Prolog.

Keywords: parallel processing, Concurrent Prolog, logic programming, graph reduction, dataflow, systolic algorithms, Turtle geometry.

How to evaluate proposed architectures?

One criteria [Arvind]: Scalability

- For twice the money, get twice the computer,

  or

- The architecture should remain feasible as the number of processors goes to infinity.

Implication of scalability:

Non-uniform communication/memory-reference costs (Crossbar switches and their approximations are not scalable).

Implication of non-uniform communication costs:

Ensuring the locality of communication and memory-references is crucial for efficient parallel processing.

The Basic Question:

- How to control communication?

or

- How to ensure locality of communication?

Several answers:

(1) "Smart" load-balancing algorithms.

(2) "Smart" compilers.

(3) Smart programmers and algorithm designers.

Statement:

Answer (1) is not feasible.

A notation for specifying process-to-processor mapping is required by answers (2) and (3).

Statement:

Designing efficient process structures (which localize communication) for parallel program is as difficult as designing efficient data-structures for a sequential program (cf. systolic algorithms).

Statement:

No practical system for selecting data-structures is available.

Conclusion:

Answer (2) is not feasible (in the forseeable future).

Systolic Algorithms

- Developed by Kung and colleagues at CMU, for direct implementation in VLSI.

- Combine pipelining and multiprocessing in a single framework.

- Achieve massive parallelism

- Applied so far mostly to numeric problems.

Example: A systolic algorithm
for band-matrix multiplication

(See figure).

Statement:

Multiprocessing adds a new dimension of programming.

To run efficiently on a sequential computer, programs must control the usage of

- Space and

- Time.

To run efficiently on a parallel computer, programs must control the usage of

- Space,

- Time, and

- Communication.

Implication:

Algorithm designers and programmers should have control over the mapping of processes to processors.

Implication:

A programming language for a parallel computer should include a mapping notation.

Implication:

The multiprocessor's interconnection scheme must be simple, intuitive, and of general purpose, for programmers to use it effectively.

Objection:

Programming is difficult as is. Incorporating explicit control of process-to-processor mapping would make it horrendous.

Answer:

Programming in higher-level languages is easier.

There seems to be a tradeoff between the complexity of data-structures in sequential programs and communication-structures in parallel programs, so overall program complexity is preserved.

Our experience shows that the mapping component of a program is relatively small, is not difficult to specify (children's programming...), and can be debugged independently of the main algorithm.

Objection:

Explicit mapping by user is too simple-minded and rigid. More sophisticated and flexible mapping strategies are essential for many applications.

Answers:

- Simple is beautiful.

- Our example programs provide evidence to the contrary.

- Load-balancing algorithms may be required in a multi-user environment. However, they should be implemented by the systems hacker, not the hardware architect. Hence the kernel programming language needs a mapping notation, Q.E.D.

A Concurrent Prolog Ad

Concurrent Prolog combines the logic programming computation model with guarded-command indeterminacy and dataflow synchronization.

It is simple. It adds to pure logic programs only
two control primitives:

- The commit operator (for indeterminacy).
- Read-only annotations (for synchronization).

It is expressive. Applied so far to:

- Systems programming
  (ICOT TR-003; POPL-84; LPS-84;
  A.Takeuchi and K.Furukawa, LPW-83).

- Systolic algorithms (this talk).

- Distributed algorithms
  (A.Shafrir, Weizmann TR CS83-12; L.Hellerstein, LPS-84).

- Object-oriented programming and constraint systems
  (A.Takeuchi, J. New Generation Computing 1(1)).

- Parallel parsing
  (H.Hirakawa, ICOT TR-008).

- Hardware specification and debugging
  (N.Suzuki, in Logic Programming and its
  Applications, Warren & van Caneghem (eds), 1984).

- Implementation of embedded languages:
  Or-parallel Prolog (Hirakawa et al., ICOT TR-020).
  Mandala: A knowledge-programming language
  (K.Furukawa, A.Takeuchi, and S.Kunifuji, ICOT TR-029).

It is amenable to efficient implementation (we hope...).
So far has only an interpreter implemented in
Prolog (ICOT TR-003).

The Bagel: A Systolic Concurrent Prolog Machine

- Architecture:

  rectangular grid of transputers with
  nearest-neighbor and shifted end-round
  torroidal interconnections.

- Programming language:

  Concurrent Prolog, augmented with Turtle
  programs as a mapping notation.

- Major application method:

  Systolic algorithms.

- Implementation state:

  Software simulators written in Prolog (with Turtle graphics) and Concurrent Prolog (very slow) exist.
  Programs below were debugged using the Turtle graphics simulator.

Constructing the Bagel: Step 1

(See figure).

Constructing the Bagel: Step 2

(See figure).

The Bagel

(See figure).

Aspects of the Bagel's interconnection scheme

- Virtual infinite two dimensional grid (programs need not know the dimensions of the Bagel). (convenient communication structure for many applications)
- A path in any direction will visit every processor once, before returning to its origin. (supports even mapping).
- Simple to implement in current and forthcoming technologies.
- Scalable.

Aspects of the Bagel's computation model

- Basic computation step:

  Process reduction.
- Synchronization:

  Data-flow (read-only variables).

- Interprocessor communication:

  Packets containing:

  * instantiations of shared variables.
  * processes and their associated programs.
  * process control messages (success, failure).

Aspects of the Bagel's transputer

Each transputer consists of:

- Reduction processor (pipelined?)
- Random access memory
- Communication processor.
- Associative memory
- Interface to external I/O.

Possible optimizations:

- Cache
- Two-port memory.

Possible approximations (hardware simulators):

- Reduction processor and communication processor are off-the-shelf chips.
- Associative memory simulated by a hash table.

Schematic design of the Bagel's transputer

(See Figure)

The Bagel's kernel language

Concurrent Prolog augmented with fixed-instruction Turtle programs as notation for mapping processes to processors.

Goals (processes) can be of the form Goal@TP meaning, solve the goal (execute the process) Goal at the processor specified by the Turtle program TP.

Each process, like a Turtle, has a position and a heading. The initial position and heading of a child process is inherited from its parent.

Fixed-instruction Turtle programs are a sequence of instructions of the form:

forward(Distance), back(Distance), left(Angle), right(Angle), turn(Degree) (absolute heading) [i,j] (absolute position), stay (no-op).

(currently only integer Distances and 90 degree Angles are implemented).

Examples of process configuration schemes

Linear pipes:

- Sieve of Erasthotenes.
- Bubble sort.
- Merge sort.
- Natural language interface to a DB (scheme).

Rectangular arrays:

- Rectangular matrix multiplication.
- Band-matrix multiplication.
- Dynamic programming.
- Array relaxation.

Static H-trees:

- Divide-and-conquer (towers of Hanoi).

Dynamic H-trees:

- Distributed database (scheme).

Linear pipe: Sieve of Erasthotenes

Abstract process structure:

```
primes :- integers, sift@forward.

sift :- filter, sift@forward.

filter :-  filter.

integers :- integers.
```

Concurrent Prolog Code:

```
primes(J) :- integers(2,I), sift(I?,J)@forward.

sift([P|I],[P|R1?]) :- filter(I,P,R), sift(R?,R1)@forward.

filter([N|I],P,R) :-  0=:=N mod P | filter(I,P,R).
filter([N|I],P,[N|R?]) :- 0=\=N mod P | filter(I,P,R ).

integers(N,[N|I?]) :- N1:=N+1, integers(N1,I).
```

Linear pipe: Bubble-sort

(Linear time and process complexity)

Abstract process structure:

```
bsort :-
    bfilter,
    bsort@forward.
bsort.

bfilter :-
    bfilter.
bfilter.
```

Concurrent Prolog Code:

```
bsort([X|Xs], [Y|Ys]) :-
    bfilter(X, Xs?, Xs1, Y),
    bsort(Xs1?, Ys)@forward.
bsort([], []).

bfilter(X1, [X2|Xs], [X2|Ys], Y) :-
    X1<X2 | bfilter(X1, Xs?, Ys, Y).
bfilter(X1, [X2|Xs], [X1|Ys], Y) :-
    X1>=X2 | bfilter(X2, Xs?, Ys, Y).
bfilter(X, [], [], X).
```

Linear pipe: merge-sort

(Linear time and logarithmic process complexity)

Abstract process structure:

```
msort.
msort :-
    merge_all,
    msort@forward.

merge_all.
```

```
merge_all :-
    merge2,
    merge_all.

merge2.
merge2 :-
    merge2.
```

Linear pipe: merge-sort

Concurrent Prolog Code:
(Note: input is a list of sorted lists).

```
msort([], []).
msort([X], X).
msort(Xs, Zs) :-
    Xs\=[], Xs\=[_] |
    merge_all(Xs, Ys),
    msort(Ys?, Zs)@forward.

merge_all([], []).
merge_all([X], [X]).
merge_all([X1,X2|Xs], [Y?|Ys?]) :-
    merge2(X1?, X2?, Y),
    merge_all(Xs, Ys).

merge2([], X, X).
merge2(X, [], X).
merge2([X|Xs], [Y|Ys], [X|Zs?]) :-
    X=<Y | merge2(Xs, [Y|Ys], Zs).
merge2([X|Xs], [Y|Ys], [Y|Zs?]) :-
    X>Y | merge2([X|Xs], Ys, Zs).
```

Linear pipe: Natural language interface to a database (scheme).

Abstract process structure:

```
process :-
    morphological,
    syntax@forward(1),
    semantics@forward(2),
    pragmatics@forward(3),
    planning@forward(4).
```

Concurrent Prolog code (scheme):

```
process(String, Query) :-
    morphological(String?, Tokens),
    syntax(Tokens, SyntaxTree)@forward(1),
```

```
semantics(SyntaxTree?, Formula)@forward(2),
pragmatics(Formula?, Formula1)@forward(3),
planning(Formula1?, Query)@forward(4).
```

Advantages:

- Can pipelined multiple queries.
- Code for each stage resides only in one processor.

Rectangular array: matrix multiplication (1)

(Linear time and quadratic process complexity)

Abstract process structure:

```
mm.
mm :- vm@right, mm@forward.

vm.
vm :- ip, vm@forward.

ip :- ip.
ip.
```

Concurrent Prolog code:

```
mm([], _, []).
mm([X|Xs], Ys, [Z|Zs]) :-
    vm(X, Ys?, Z)@right,  mm(Xs?, Ys, Zs)@forward.

vm(_, [], []).
vm(Xs, [Y|Ys], [Z|Zs]) :-
    ip(Xs?, Y?, Z),  vm(Xs, Ys?, Zs)@forward.

ip(Xs, Ys, Z) :-
    ip(Xs, Ys, 0, Z).

ip([X|Xs], [Y|Ys], Z0, Z) :-
    Z1:=(X * Y)+Z0,  ip(Xs?, Ys?, Z1, Z).
ip([], [], Z, Z).
```

Rectangular array: matrix multiplication (2)

A variant of the previous program, that pipelines the vectors, instead of sending them as a whole.

```
mm([], _, []).
mm([X|Xs], Ys, [Z|Zs]) :-
    vm(X, Ys?, Ys1, Z)@right,  mm(Xs?, Ys1?, Zs)@forward.

vm(_, [], [], []).
vm(Xs, [Y|Ys], [Y1|Ys1], [Z|Zs]) :-
```

```
    ip(Xs?, Xsl, Y?, Yl, Z),  vm(Xsl?, Ys?, Ysl, Zs)@forward.

ip(Xs, Xsl, Ys, Ysl, Z) :-
    ip(Xs, Xsl, Ys, Ysl, 0, Z).

ip([X|Xs], [X|Xsl], [Y|Ys], [Y|Ysl], Z0, Z) :-
    Z1:=(X * Y)+Z0,  ip(Xs?, Xsl, Ys?, Ysl, Z1, Z).
ip([], [], [], [], Z, Z).
```

Rectangular array: dynamic programming

(The systolic algorithm of Kung, Guibas, and Thompson)

Abstract process structure:

```
table.
table :-
    row@right,
    table@forward.

row.
row :-
    entry,
    row@forward.

entry.
```

Rectangular array: dynamic programming

Concurrent Prolog code:

Input: a list of triples (0,D1,D2), where the D's are matrix dimensions.

Output: (W,D1,D2), where W is the number of multiplications in optimal parenthesization.

```
table([W],W).
table(Ws,Min) :-
    Ws\=[_] |
    row(Ws,Wsl)@right,
    table(Wsl?,Min)@forward.

row([_],[]).
row([W1,W2|Ws],[W|Wsl?]) :-
    entry(W1,W2,W),
    row([W2|Ws],Wsl)@forward.

entry((W1,L1,R1),(W2,L2,R2),(W,L1,R2)) :-
    W:=min(W1+L1*R1*R2, W2+L1*L2*R2).
```

Note how diagonal communication channels between table entries are created by the row procedure.

H-trees: A scheme for divide-and-conquer

Abstract process structure:

```
htree.
htree :-
    htree@(left,forward),
    htree@(right,forward).
```

Concurrent Prolog code:

```
htree(0).
htree(D++) :-
    htree(D)@(left,forward(2(D/2))),
    htree(D)@(right,forward(2(D/2))).
```

Note: p(X++,...) :- ...
Is a shorthand for: p(Xl,...) :- Xl>0 | X:=Xl-1, ...

H-trees: The Towers of Hanoi

Abstract process structure:

```
hanoi.
hanoi :-
    free,
    hanoi@(left,forward),
    hanoi@(right,forward).

free.
```

Concurrent Prolog code:

```
hanoi(0,From,To,(From,To)).
hanoi(N++,From,To,(Before,(From,To),After)):-
    free(From,To,Free),
    hanoi(N,From,Free,Before)@(left,forward(2(N/2))),
    hanoi(N,Free,To,After)@(right,forward(2(N/2))).

    free(a,b,c).
    free(a,c,b).
    free(b,a,c).
    free(b,c,a).
    free(c,a,b).
    free(c,b,a).
```

Rectangular array: band-matrix multiplication

(The systolic algorithm of Kung and Leiserson, linear time and quadratic process complexity)

Abstract process structure:

```
mm :-
     spawn_isp,
     arm@forward,
     arm@right,
     mm@(forward,right,forward,left).
mm.

arm.
arm :-
     spawn_isp,
     arm@forward.

spawn_isp :-
     isp.
spawn_isp :-
     forward,
     isp.

forward.
forward :-
     forward.

isp :-
     isp.
isp.
```

Concurrent Prolog code:

```
mm(D,[Ain|Asin],[Bin|Bsin],c(Clout,Cout,Crout)) :-
     spawn_isp(D,0,Cin?,Cout,Ain?,Aout,Bin?,Bout),
     arm(D,Asin?,Asout,Clin,Clout,Bout)@forward,
     arm(D,Bsin?,Bsout,Crin,Crout,Aout)@right,
     mm(D-1,Asout?,Bsout?,c(Clin,Cin,Crin))@(forward,right,forward,left).
mm(D,[],[],c([],[],[])).

% D is the diagonal distance from the center-point x.
% V is the vertical (horizontal) distance from x's diagonal.
arm(D,Asin,Asout,Cin,Cout,Bin) :-
    arm(D,1,Asin,Asout,Cin,Cout,Bin).

arm(D,V,[],[],[],[[]],_).
arm(D,V,[Ain|Asin],[Aout|Asout],[Cin|Csin],[Cout|Csout],Bin) :-
    spawn_isp(D,V,Cin?,Cout,Ain?,Aout,Bin?,Bout),
    V1:=V+1,
    arm(D,V1,Asin?,Asout,Csin,Csout,Bout)@forward.
```

```
spawn_isp(0,V,Cin,Cout,Ain,Aout,Bin,Bout) :-
    % we are in the 0's area...
    isp(Cin,Cout,Ain,Aout,Bin,Bout).
spawn_isp(D,V,Cin,Cout,Ain,Aout,Bin,Bout) :-
    D>0 | % we are in the A (or B) area...
    forward(min(D,V),Ain,Ain1,Aout,Aout1),
    isp(Cin,Cout,Ain1?,Aout1,Bin,Bout).
spawn_isp(D,V,Cin,Cout,Ain,Aout,Bin,Bout) :-
    D<0 | % we are in the C area...
    forward(-D,Cin?,Cin1,Cout,Cout1),
    isp(Cin1?,Cout1,Ain,Aout,Bin,Bout).

forward(0,Cin,Cin,Cout,Cout).
forward(N++,Cin,Cin2,Cout,Cout2) :-
    get_c(C,Cin,Cin1), send(C,Cout,Cout1),
    forward(N,Cin1?,Cin2,Cout1,Cout2).

get_c(0,[],[]).
get_c(C,[C|Cs],Cs).

isp(Cin,[C1|Cout],[A|Ain],[A|Aout],[B|Bin],[B|Bout]) :-
    get_c(C,Cin,Cin1),
    C1:=C+(A * B) |
    isp(Cin1?,Cout,Ain?,Aout,Bin?,Bout).
isp(Cs,Cs,As,As,[],[]).
isp(Cs,Cs,[],[],Bs,Bs).
```

Rectangular array: array relaxation

Abstract process structure (simplified):

```
relax :-
    monitor,
    matrix@forward.

matrix :-
    vector@right,
    matrix@forward,
    merge_monitor.

vector :-
    spawn_cell,
    vector@forward,
    merge_monitor.

spawn_cell :-
    cell_monitor,
    cell.

cell :-
    cell.
cell_monitor :-
    cell_monitor.

merge_monitor :-
    merge_monitor.

monitor :-
```

monitor.

Rectangular array: array relaxation

Concurrent Prolog code:

```
relax(X,Y) :-
     monitor(Monitor?,Halt),
     matrix((1,1),X,Bottom,Top,Monitor,Halt?)@forward,
     tie_vector(Bottom?),
     tie_vector(Top?).


%matrix(Coordinates,Matrix,Nextrow channels,Final channels).
matrix(_,[],Top,Top,[],_).
matrix((I,J),[X|Xs],Bottom,Top,Monitor,Halt) :-
     vector((I,J),X,Bottom,Bottom1,Vmonitor,Halt)@right,
     matrix((I+1,J),Xs,Bottom1,Top,Mmonitor,Halt)@forward,
     merge_monitor(Mmonitor?,Vmonitor?,Monitor).

tie_vector([]).
tie_vector([X|Xs]) :-
     tie_cell(X), tie_vector(Xs?).

vector(IJ,Xs,Bottom,Top,Monitor,Halt) :-
     vector(IJ,Xs,Left,Right,Bottom,Top,Monitor,Halt),
     tie_cell(Left),
     tie_cell(Right).

%vector(Coord,Leftchannel,Rightchannel,Bottomchannels,Topchnls).
vector(IJ,[],Right,Right,[],[],[],_).
vector((I,J),[X|Xs],Left,Right,[Bottom|Bs],[Top|Ts],Monitor,Halt) :-
     spawn_cell((I,J),X,Left,Left1,Bottom,Top,Cmonitor,Halt),
     vector((I,J+1),Xs,Left1,Right,Bs,Ts,Vmonitor,Halt)@forward,
     merge_monitor(Vmonitor?,Cmonitor?,Monitor).

tie_cell(c(X,X)).

spawn_cell(IJ,X,c(Lin,Out),c(Out,Rin),c(Bin,Out),c(Out,Tin),Monitor,Halt) :-
     send(X,Out,Out1) |
     cell_monitor(X,Out1?,Monitor),
     cell(IJ,Halt,Out1,Lin?,Rin?,Bin?,Tin?).

cell(IJ,halt,[],_,_,_,_).
cell(IJ,Halt,Out,[X1|L],[X2|R],[X3|B],[X4|T]) :-
     X := ((X1+X2+X3+X4) / 4) |
     send(X,Out,Out1),
     cell(IJ,Halt,Out1,L?,R?,B?,T?).

cell_monitor(X1,[X2|Xs],[halt|Ys]) :-
    X1=:=X2 | cell_monitor(X2,Xs?,Ys).
cell_monitor(X1,[X2|Xs],[continue|Ys]) :-
    X1\=X2 | cell_monitor(X2,Xs?,Ys).
cell_monitor(_,[],[]).

merge_monitor(Xs,Ys,Zs,Halt) :-
```

```
    merge_monitor(Halt,continue,Xs,Ys,Zs).

merge_monitor(halt,_,_,_,[]).
merge_monitor(Monitor,State,[X|Xs],[Y|Ys],Zs) :-
    merge_messages(State,X,Y,Zs,State1,Zs1),
    merge_monitor(Monitor,State1,Xs?,Ys?,Zs1).
merge_monitor(Monitor,State,Xs,[],Xs).
merge_monitor(Monitor,State,[],Xs,Xs).

merge_messages(halt,continue,Y,[continue|Zs],continue,Zs).
merge_messages(halt,X,continue,[continue|Zs],continue,Zs).
merge_messages(X,halt,halt,[halt|Zs],halt,Zs).
merge_messages(continue,X,continue,Zs,continue,Zs).
merge_messages(continue,continue,Y,Zs,continue,Zs).

monitor([halt|Xs],halt).
monitor([continue|Xs],Monitor) :-
    monitor(Xs?,Monitor).
```

Chaotic process: A Turtle.

Concurrent Prolog code:

```
turtle :-
    instream(X), turtle(X?).

turtle([]).
turtle([X|Xs]) :- turtle(Xs?)@X.
```

Dynamic H-trees; a scheme for a distributed database

- Relations are stored in the leaves.
- Tree nodes route queries and merge responses.
- Database grows dynamically.

Abstract process structure:

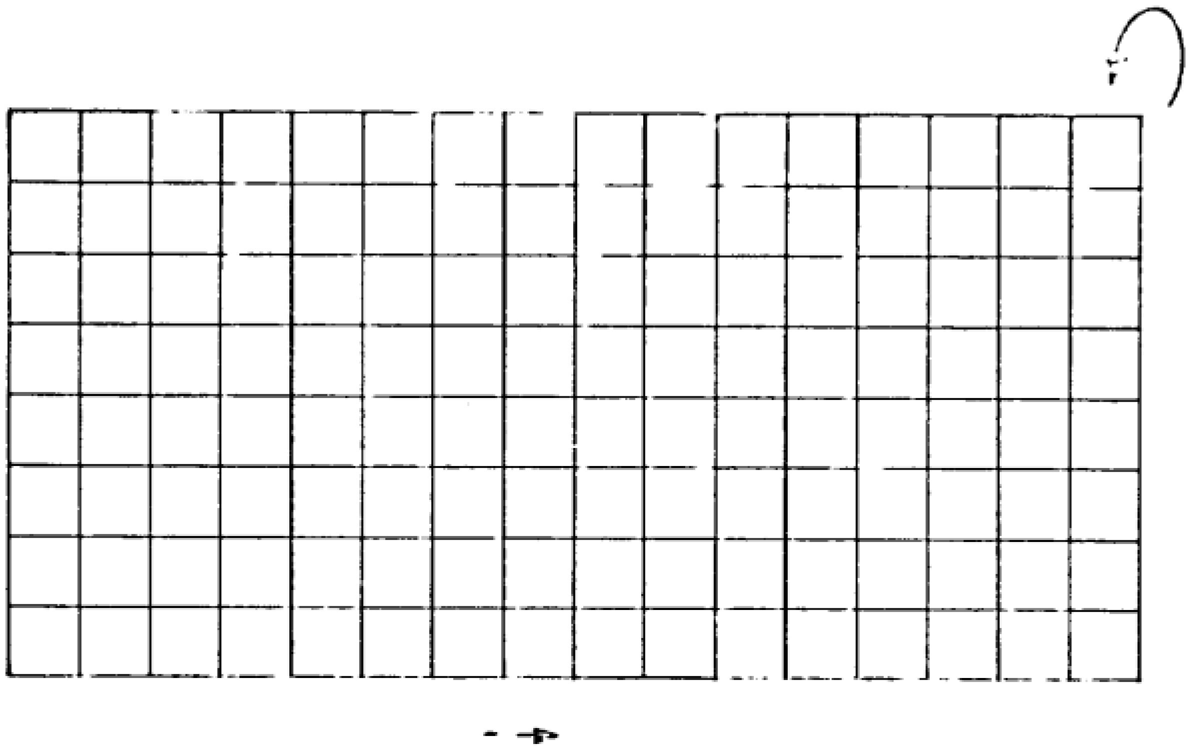
```
root :-
    root,
    leaf.

root :-
    root.


tree :-
    odd |
    tree@TP.
```

```
tree:-
      even |
      tree@TP.
tree.

leaf :- leaf_split@TP.
leaf.

leaf_split :-
      tree,
      leaf@(right,forward),
      leaf@(right,forward).
```

Concurrent Prolog code:

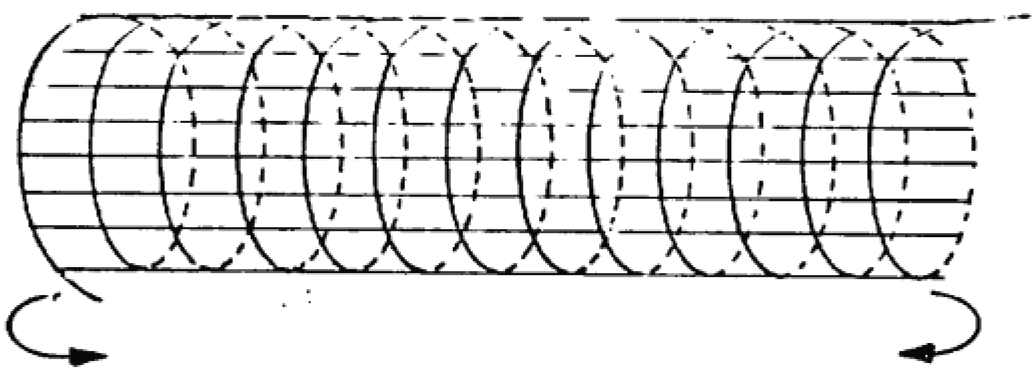
```
root(Xs) :-
      root(0,Xs?,Ys),
      leaf(Ys?).

root(D,[split|Xs],[split(D1,stay)|Ys?]) :-
      D1:=D+1,
      root(D1,Xs?,Ys).
root(D,[],[]).

tree([split(D++,TP)|Xs],
       [split(D,(right,TP,left))|L?],
       [split(D,(left,TP,right))|R?]) :-
      odd(D) |
      tree(Xs?,L,R)@TP.
tree([split(D++,TP)|Xs],
       [split(D,(right,TP,left,forward(D1)))|L?],
       [split(D,(left,TP,right,forward(D1)))|R?]) :-
      even(D), D1:=(2(D/2-1)) |
      tree(Xs?,L,R)@TP.
tree([],[],[]).

leaf([split(_,TP)|Xs]) :- leaf_split(Xs?)@TP.
leaf([]).

leaf_split(Xs) :-
      tree(Xs,L,R),
      leaf(L?)@(left,forward),
      leaf(R?)@(right,forward).

odd(X) :- X=\=(X/2)*2.
even(X) :- X=:=(X/2)*2.
```

Constructing the Bagel: Step 1



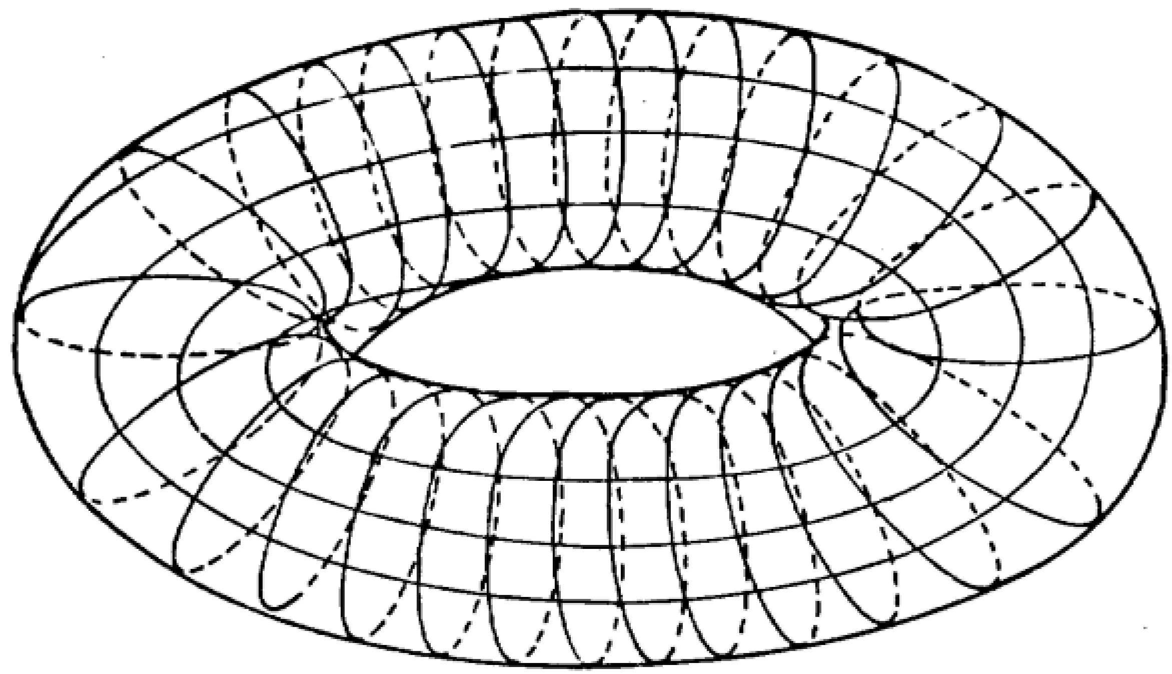
Shift 1

Constructing the Bagel: Step 2

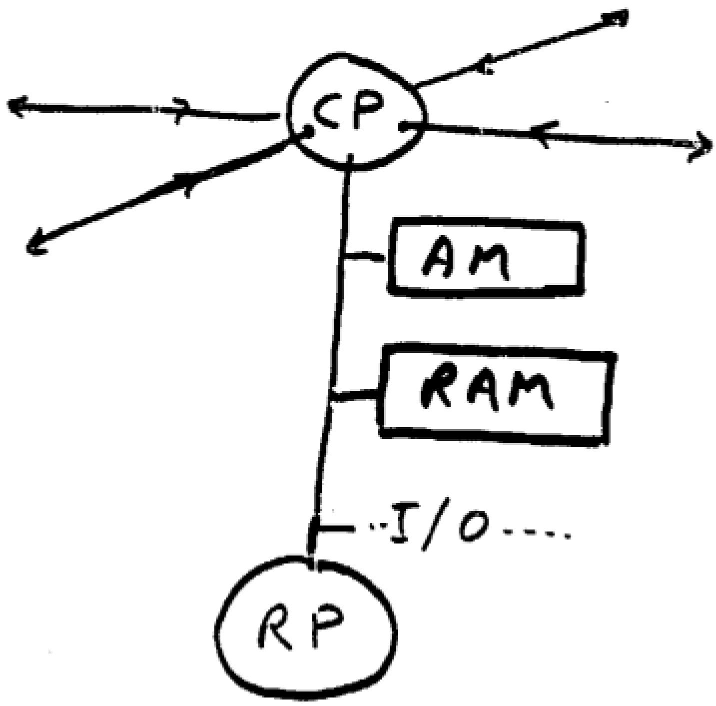


Shift 1

The Bagel

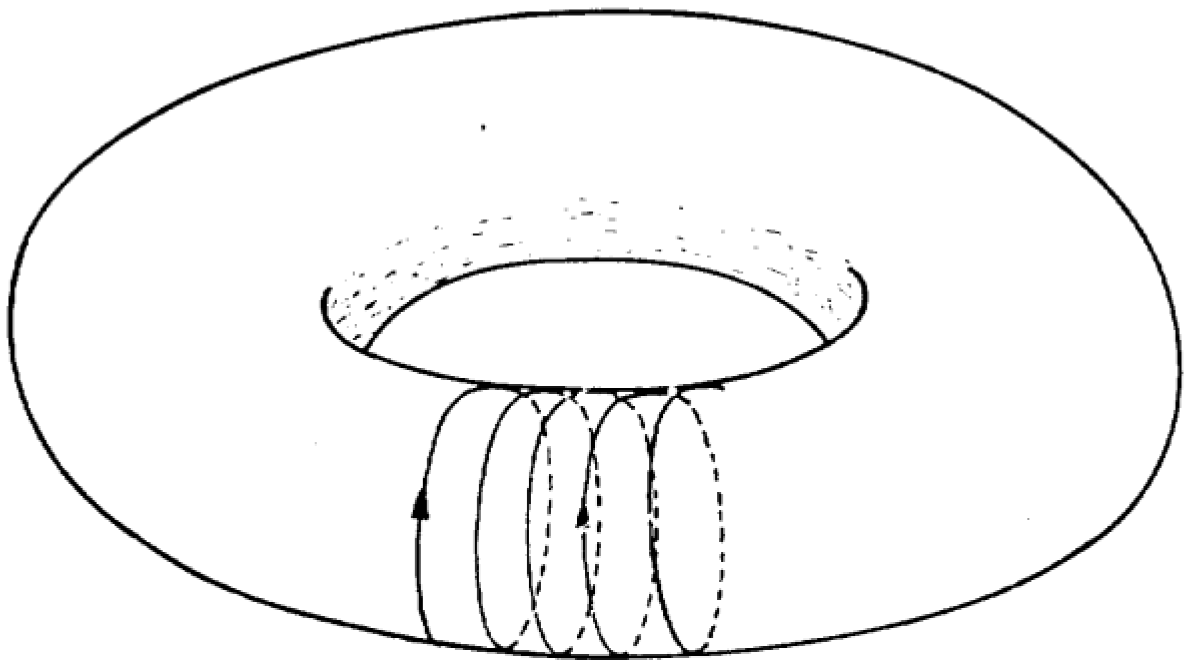


Schematic design of the Bagel's transputer.



CP - communication processor
RP - reduction processor
AM - associative memory
RAM - random access memory

Spawning a vector of processes

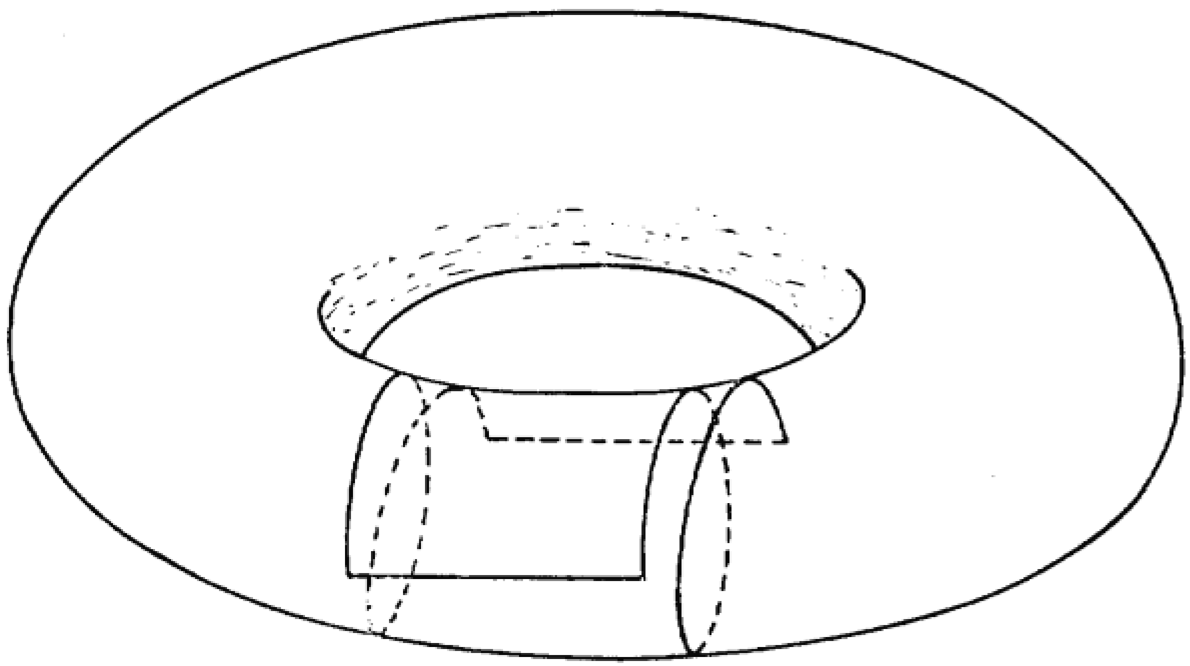


```
-(filter (23 . ?r?) 13 (23
-(filter (23 . ?r?) 11 (23
-(filter (23 . ?r?) 7 (23
-(filter (25 . ?r?) 5 ?r)
-(filter (25 . ?r?) 3 (25          ↑(sift ?r? ?r1)
-(filter (27 . ?i?) 2 (27        -(sift (23 . ?r?) (23 . ?r
-(integers 27 (27 . ?i?))        -(filter (23 . ?r?) 19 (23
x(prolog (format t "*** ou       -(filter (23 . ?r?) 17 (23

Turtle
*** outstream: 3
*** outstream: 5
*** outstream: 7
*** outstream: 11
*** outstream: 13
*** outstream: 17
*** outstream: 19
*** outstream: 23
>Breakpoint BREAK; Resume to continue, Abort to quit.

CS is not defined in the current universe, but is a Lisp function,
Run CS in Lisp? (y, n, or g) Yes
?
OK
(define-predicate top-level-predication ((top-level-predication ?g) (cp ?g)))
OK
(primes)*** outstream: 2
Prolog level 0
11/20/83 23:02:19 Kahn          USER:          Tyi                FILE serving ICOT-LMI
```

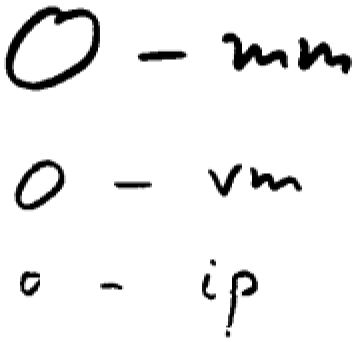
```
↑(bfilter 34 (34 . ?xs) (3

↑(bfilter 6 (34 . ?xs) (34

↑(bfilter 6 (34 . ?xs) (34

↑(bfilter 5 (34 . ?xs) (34

↑(bfilter 5 (34 . ?xs) (34

↑(bfilter 3 (34 . ?xs) (34

↑(bfilter 2 (34 246 23 43          ↑(bsort ?xs1? ?ys)

↑(out (?y . ?ys))              ↑(bsort (45 . ?xs) (?y . ?

Bagel Simulator
(btest)
>Breakpoint BREAK; Resume to continue, Abort to quit.

     GC: You have 116,145 words of consing left before (GC-ON) may fail.
     ve 7,937,194 words left if you elect not to garbage collect.
     :-STATUS) for more information.]




Prolog level 0
11/20/83 23:25:17 Kahn          USER:       Tyi          FILE serving ICOT-LM1
```

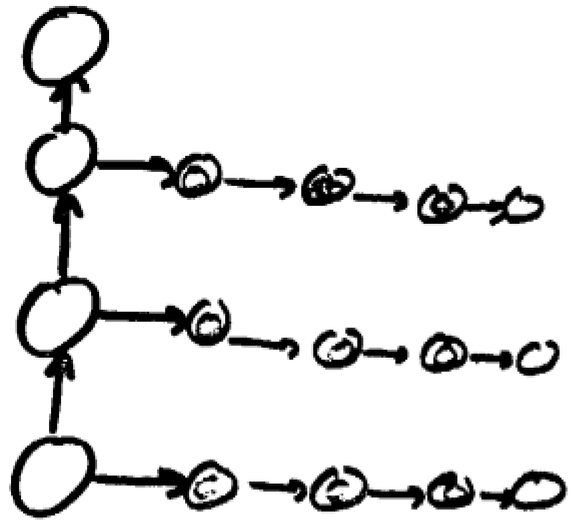
```
-(bfilter 23 nil nil 23)(5

-(bfilter 6 nil nil 6) (56          -(bsort nil nil)

-(bfilter 6 nil nil 6) (56      -(bfilter 246 nil nil 246)

-(bfilter 5 nil nil 5) (56      --(bfilter 56 nil nil 56)(

-(bfilter 5 nil nil 5) (56      --(bfilter 45 nil nil 45)5

-(bfilter 3 nil nil 3) (56      --(bfilter 43 nil nil 43)5

-(bfilter 2 nil nil 2) ?ys      --(bfilter 34 nil nil 34)5

x(prolog (format t "*** ou      --(bfilter 34 nil nil 34)5

Bagel Simulator
(btest)*** outstream: 2
*** outstream: 3
*** outstream: 5
*** outstream: 5
*** outstream: 6
*** outstream: 6
*** outstream: 23
*** outstream: 34
*** outstream: 34
*** outstream: 43
>Breakpoint BREAK; Resume to continue, Abort to quit.

    GC: You have 205,657 words of consing left before (GC-ON) may fail.
   ve 7,948,369 words left if you elect not to garbage collect.
   C-STATUS) for more information.]


Prolog level 0
11/20/83 23:10:46 Kahn          USER:        Ty:            FILE serving ICOT-LM1
```

Spawning on array of processes

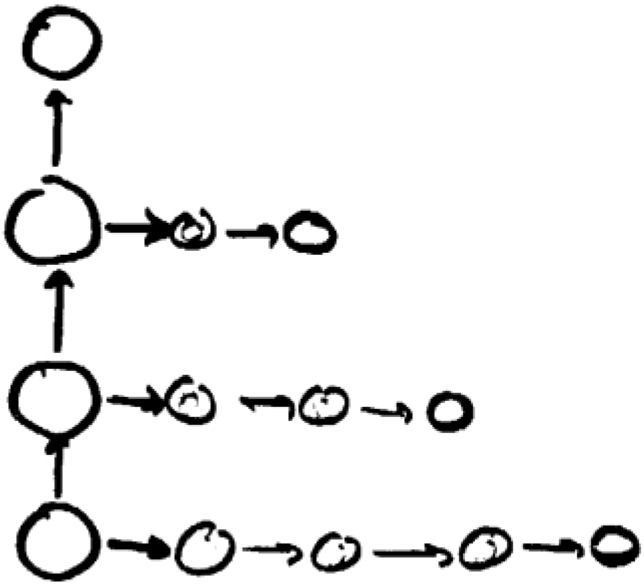


Example: Spawning the matrix multiplication processes



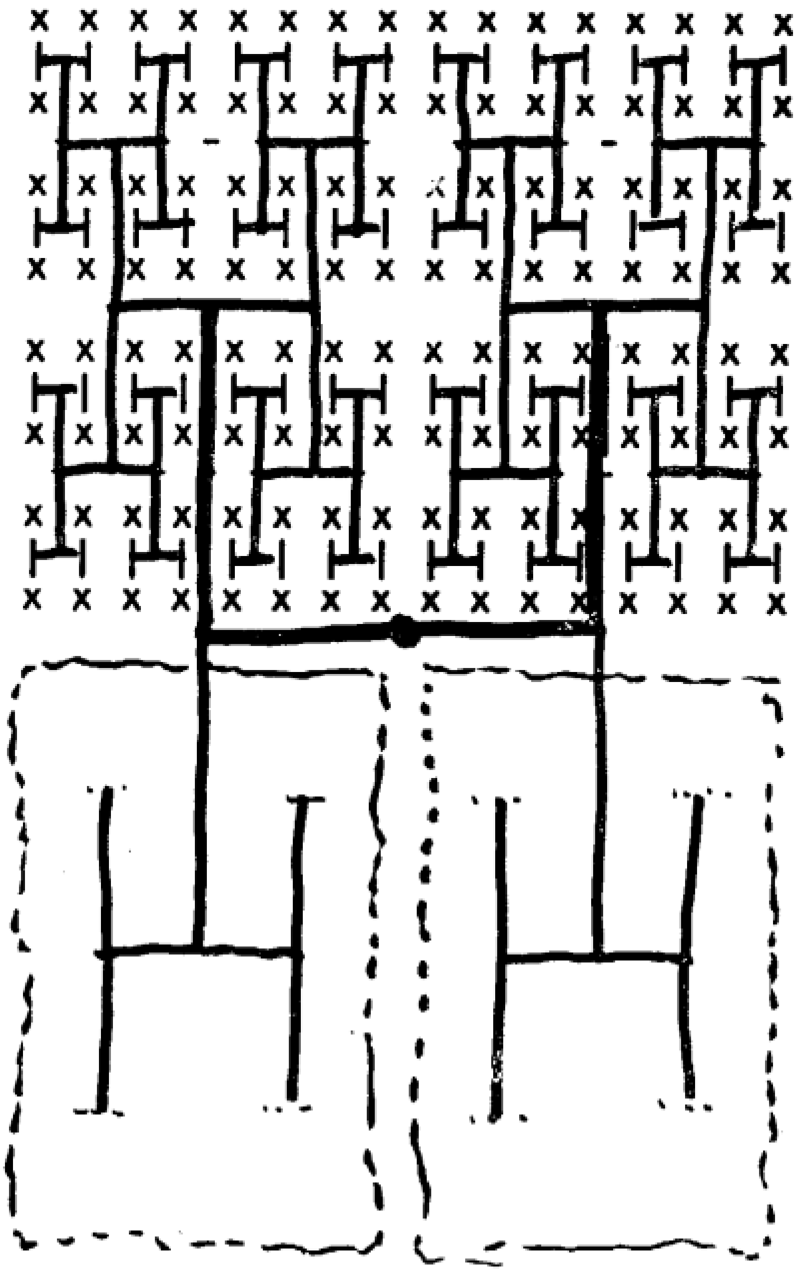


O - table
o - row
∘ - entry

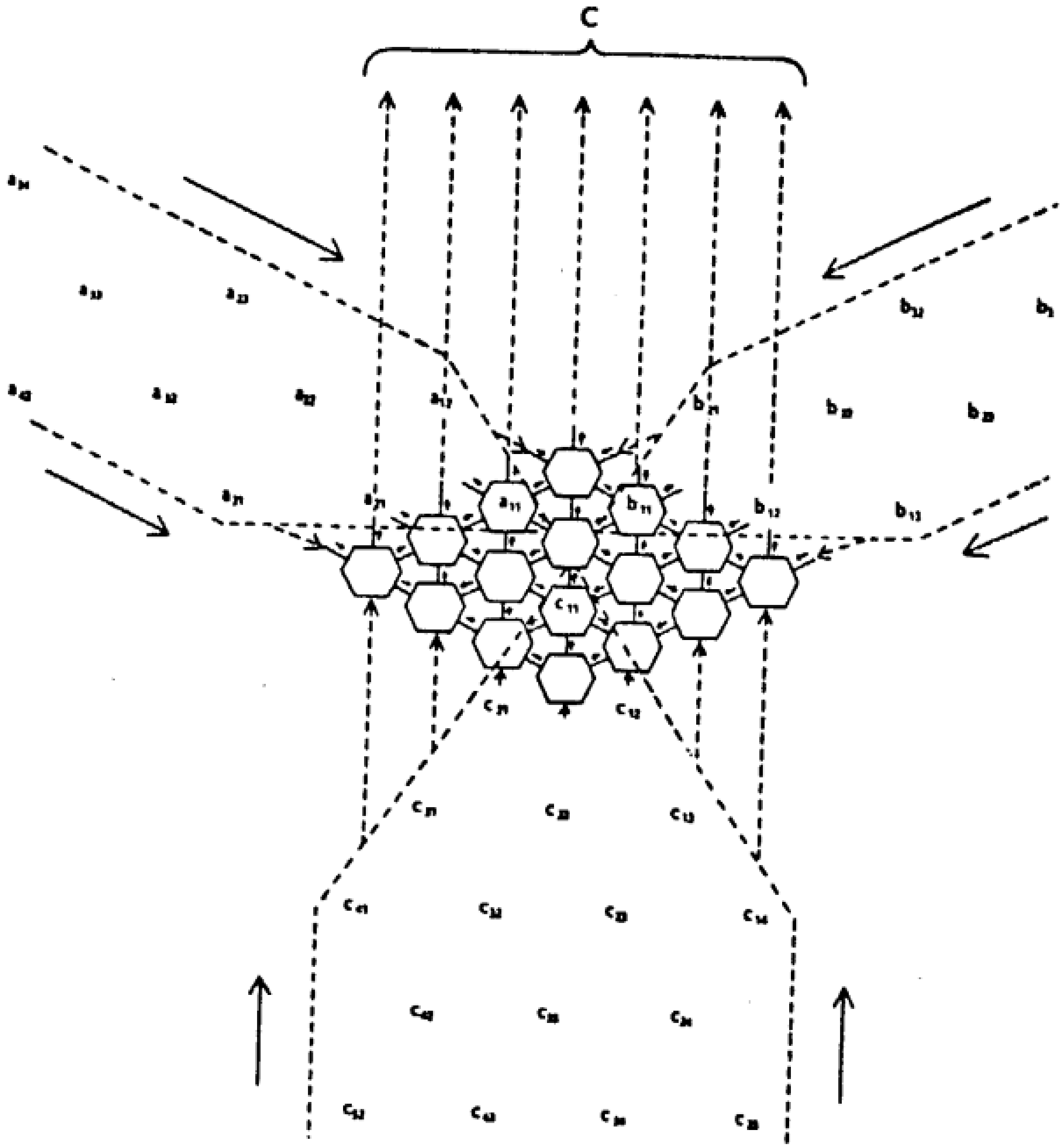


Spawning a dynamic programming table

An Example: Spawning a binary tree of 16x16=256 processes on the Bagel



Example *A systolic algorithm for band-matrix multiplication*



$$
\begin{bmatrix}
a_{11} & a_{12} & & & 0 \\
a_{21} & a_{22} & a_{23} & & \\
a_{31} & a_{32} & a_{33} & a_{34} & \\
 & a_{42} & & \cdot & \\
0 & & & \cdot & \cdot
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} & b_{13} & & 0 \\
b_{21} & b_{22} & b_{23} & b_{24} & \\
 & b_{32} & b_{33} & b_{34} & b_{35} \\
 & & b_{43} & \cdot & \\
0 & & & \cdot & \cdot
\end{bmatrix}
=
\begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} & 0 \\
c_{21} & c_{22} & c_{23} & c_{24} & \\
c_{31} & c_{32} & c_{33} & c_{34} & \\
c_{41} & c_{42} & & \cdot & \\
0 & & & \cdot & \cdot
\end{bmatrix}
$$

Spawning the band-matrix system

O – mm

O – spawn_arm

o – isp