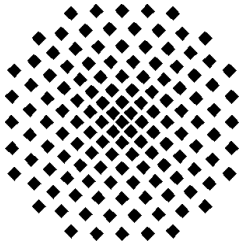
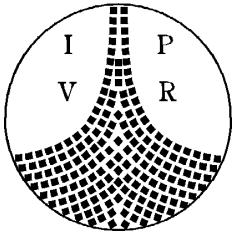




University of Stuttgart

Institute for Parallel and Distributed
High Performance Systems (IPVR)
Computer Vision Group
Breitwiesenstraße 20 – 22
D-70565 Stuttgart, Germany

PROGRAMMING IN PARALLAXIS

THOMAS BRÄUNL

Overview

- Definition of the Parallaxis programming language
- Application programs in Parallaxis

Data-Parallel Programming Language Parallaxis

History

- language definition by Thomas Bräunl, Univ. Stuttgart (Germany) in 1989
- programming environment is distributed as public domain software
- current version Parallaxis-III (1994)
- Parallaxis is widely used at universities for teaching data-parallel programming concepts

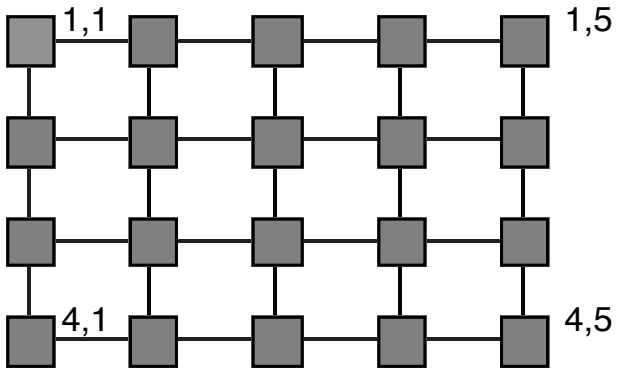
Key Features

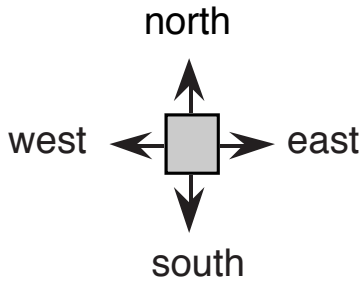
- machine independent
- based on structured sequential programming language Modula-2 (descendant of Pascal)
- each program comprises a parallel algorithm **plus** a specification of number of PEs (virtual processing elements) and connection structure (virtual topology with symbolic names)
- simulation system for single processor systems (workstations and personal computers) **and**
- parallel system on MasPar and Connection Machine
- source level debugger
- visualization tools for PE data, PE activity, and PE load

Parallel Language Concepts

Specification of Virtual Processors and Connections

Two-Dimensional Grid

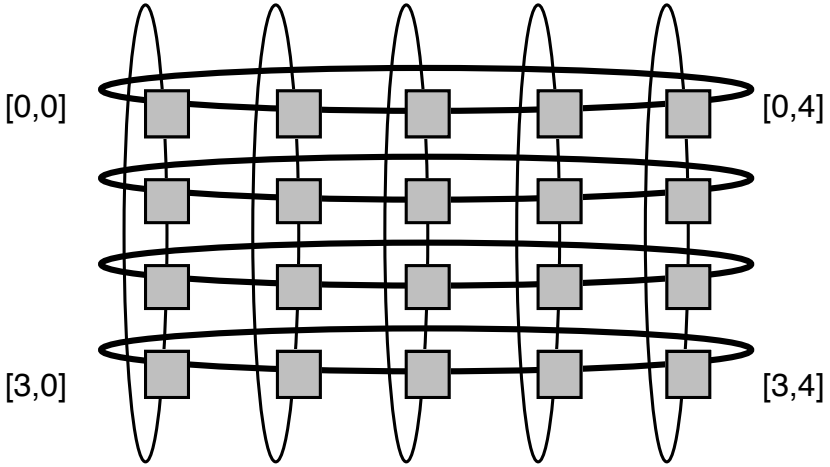





```
CONFIGURATION grid [1..4],[1..5];
CONNECTION  north: grid[i,j] →  grid[i-1, j];
            south: grid[i,j] →  grid[i+1, j];
            east : grid[i,j] →  grid[i, j+1];
            west : grid[i,j] →  grid[i, j-1];
```

Sample Topologies

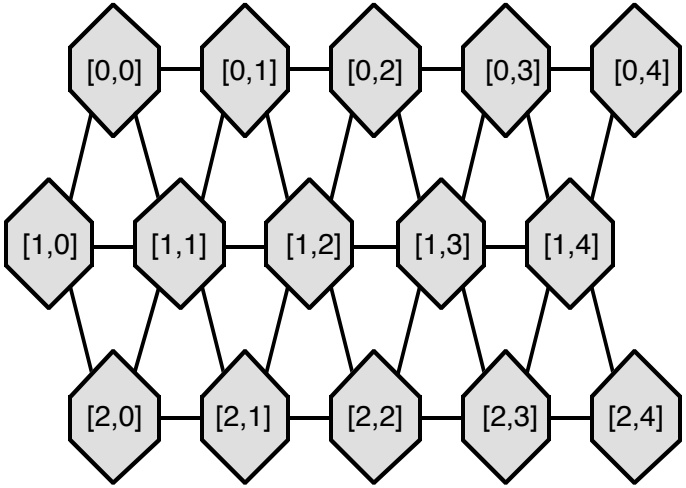
Torus




```
CONFIGURATION torus [0..h-1],[0..w-1];
CONNECTION  north: torus[i,j] →  torus[(i-1) MOD h, j];
            south: torus[i,j] →  torus[(i+1) MOD h, j];
            east : torus[i,j] →  torus[i, (j+1) MOD w];
            west : torus[i,j] →  torus[i, (j-1) MOD w];
```

(origin is upper left, `h` and `w` are constants)

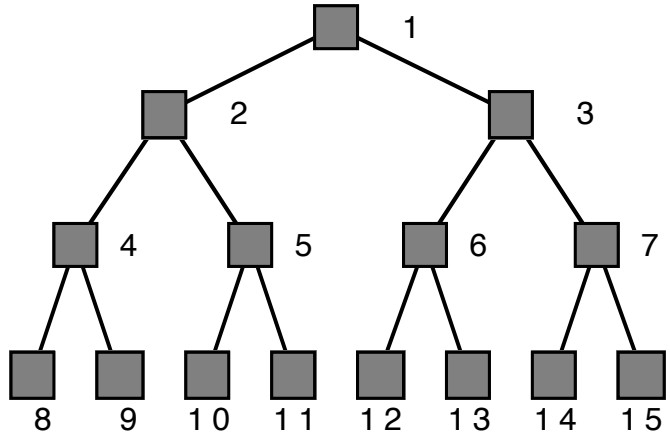
Hexagonal Grid




```
CONFIGURATION hexa [0..2],[1..4];
CONNECTION right : hexa[i,j] ↔ hexa[i  , j+1]           : left;
           up_l  : hexa[i,j] ↔ hexa[i-1, j   - i MOD 2] : down_r;
           up_r  : hexa[i,j] ↔ hexa[i-1, j+1 - i MOD 2] : down_l;
```

(double arrow denotes bi-directional connections)

Binary Tree

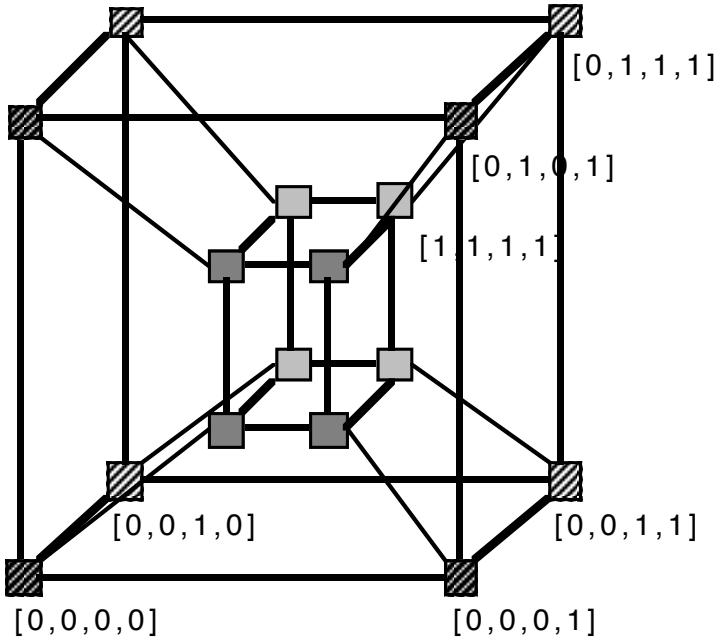



```
CONFIGURATION  tree [1..15];
CONNECTION  lchild: tree[i] →  tree[2*i];
            rchild: tree[i] →  tree[2*i+1];
            parent: tree[i] →  tree[i DIV 2];
```

An alternative using bi-directional connections is:

```
CONFIGURATION  tree [1..15];
CONNECTION lchild: tree[i] ↔ tree[2*i]      :parent;
           rchild: tree[i] ↔ tree[2*i + 1]  :parent;
```

Hypercube




```
CONFIGURATION  hyper [0..1],[0..1],[0..1],[0..1];
CONNECTION  go[1]: hyper[i,j,k,l]  →  hyper[(i+1) MOD 2, j, k, l];
            go[2]: hyper[i,j,k,l]  →  hyper[i, (j+1) MOD 2, k, l];
            go[3]: hyper[i,j,k,l]  →  hyper[i, j, (k+1) MOD 2, l];
            go[4]: hyper[i,j,k,l]  →  hyper[i, j, k, (l+1) MOD 2];
```

(square brackets in connection names denote parameterized connections)

Compound Connections

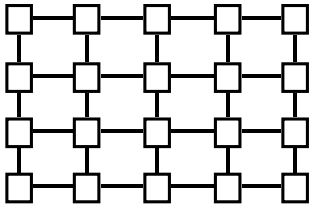

```
CONFIGURATION list [1..8];
CONNECTION next: list[i] →  {ODD (i)} list[i+1],
                            {EVEN(i)} list[i-1];
```

(curly brackets in connections denote a case distinction)

Multiple Connections

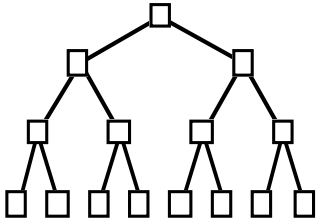
Distinct sets of PEs:

```
CONFIGURATION grid [1..200],[1..50];
...
CONFIGURATION tree [1..10000];
```



Same set of PEs:

```
CONFIGURATION grid [1..200],[1..50];
              tree [1..10000];
```



Connecting multiple configurations:

```
CONFIGURATION grid [0..199],[0..49];
              tree [1..10000];
CONNECTION mix: grid[i,j] -> tree[i*50 + j];
```

Also possible:

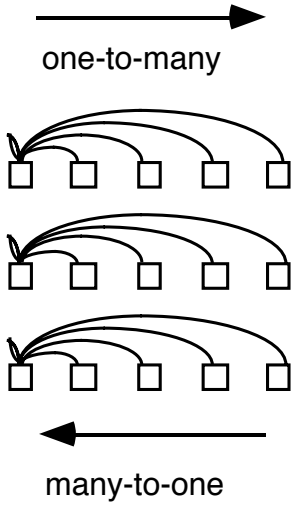
- 1:*n* connections (broadcast)

```
CONFIGURATION grid [1..100],[1..100];
CONNECTION one2many: grid[i,1] →  grid[i,1..100];
(* or abbreviated:   grid[i,1] →  grid[i,*];   *)
```

- *n*:1 connections
  (several connections of the *same name* arriving in a single PE)

```
CONFIGURATION grid [1..100],[1..100];
CONNECTION many2one: grid[i,j] →  grid[i,1];
```

- general *n*:*m* connections



Generic Connections

Replicated connection function (n is a constant)

```
CONFIGURATION hyper [0..2**n-1];
CONNECTION FOR k := 0 TO n-1 DO
             dir[k]: hyper[i] ↔ {EVEN(i DIV 2**k)} hyper[i + 2**k] :dir[k];
           END;
```

Connections without size specification

```
CONFIGURATION open_grid [*],[*];
...
CONFIGURATION small_grid = open_grid [1..100],[1..100];
```

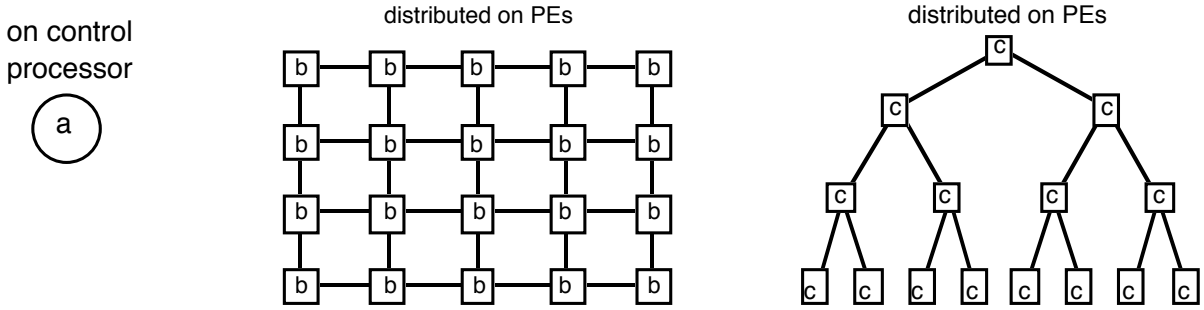
Data Declaration

basic distinction between:

scalar data allocated only once on the control processor

vector data allocated for each PE in a configuration (in its local memory)

```
VAR a: INTEGER;      (* scalar *)
    b: grid OF REAL; (* vector *)
    c: tree OF CHAR; (* vector *)
```



Procedure Parameters

- may be scalar or vector

```
PROCEDURE abc(a: INTEGER; b: grid OF INTEGER);
```

- generic parameters may be used for **any** configuration

```
PROCEDURE s_factorial(a: INTEGER): INTEGER;                          (* scalar *)
VAR b: INTEGER;
  ...
END s_factorial;


PROCEDURE v_factorial(a: VECTOR OF INTEGER): VECTOR OF INTEGER;   (* any vector *)
VAR b: VECTOR OF INTEGER;
  ...                                                                (* no data exchange *)
END v_factorial;
```

Processor Positions

```
CONFIGURATION grid [1..4],[-2..+2];
...
VAR x: grid OF INTEGER;
...
x := ID(grid);
```

1st dimension

2nd dimension

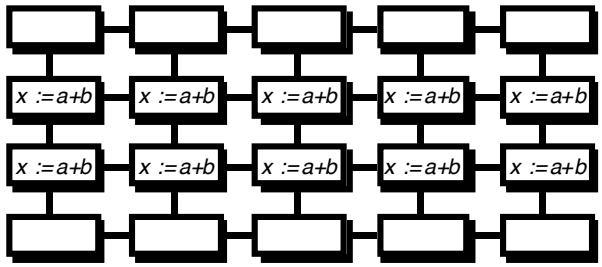
$$\texttt{ID(grid)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{pmatrix}$$

$$\texttt{DIM(grid,2)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix} \qquad \texttt{DIM(grid,1)} = \begin{pmatrix} -2 & -1 & 0 & +1 & +2 \\ -2 & -1 & 0 & +1 & +2 \\ -2 & -1 & 0 & +1 & +2 \\ -2 & -1 & 0 & +1 & +2 \end{pmatrix}$$

Other Position Functions:

`LEN(grid)`	`= 20`	returns the total number of PEs in a configuration
`LEN(grid,1)`	`= 5`	returns the size of a dimension (here dim. 1)
`LEN(grid,2)`	`= 4`	returns the size of a dimension (here dim. 2)
`RANK(grid)`	`= 2`	returns the number of dimensions
`LOWER(grid,1)`	`= -2`	returns the lower bound of a dimension (here dim. 1)
`UPPER(grid,1)`	`= 2`	returns the upper bound of a dimension (here dim. 1)
`DIM2ID(..)`		transfers DIM data to ID
`ID2DIM(..)`		transfers ID data to DIM

Parallel Execution




```
VAR x,a,b: grid OF REAL;
...
IF DIM(grid,2) IN {2,3} THEN
  x := a+b
END;
```

Parallel Selection

```
VAR x: grid OF INTEGER;
...
IF x>5 THEN x := x - 3
       ELSE x := 2 * x
END;
```

PE-ID:	*1*	*2*	*3*	*4*	*5*	
initial values of `x`:	10	4	17	1	20	
starting *then*-branch:	10	–	17	–	20	(‘–’ means inactive)
after *then*-branch:	7	–	14	–	17	
starting *else*-branch:	–	4	–	1	–	
after *else*-branch:	–	8	–	2	–	
selection done						
after *if*-selection:	7	8	14	2	17	

Parallel Iteration

```
VAR x: grid OF INTEGER;
...
WHILE x>5 DO
  x:= x DIV 2;
END;
```

PE-ID:	*1*	*2*	*3*	*4*	*5*	
initial values of `x`:	10	4	17	1	20	
starting 1st iteration:	10	–	17	–	20	(‘–’ means inactive)
after 1st iteration:	5	–	8	–	10	
starting 2nd iteration:	–	–	8	–	10	
after 2nd iteration:	–	–	4	–	5	
starting 3rd iteration:	–	–	–	–	–	
loop terminates						
after loop:	5	4	4	1	5	

Parallel Control Structures

Implicit selection or iteration statements with vector conditions

```
IF       .. THEN .. ELSE  .. END;
IF       .. THEN .. ELSIF .. THEN  .. ELSE .. END;
CASE     .. OF   .. ELSE  .. END;
WHILE    .. DO   .. END;
REPEAT   .. UNTIL;
FOR      .. TO   .. DO    .. END;
FOR      .. TO   .. BY    .. DO    .. END;
LOOP OF  .. DO   .. END;
```

with configuration name and containing: `EXIT` (usually within `IF`-selection)

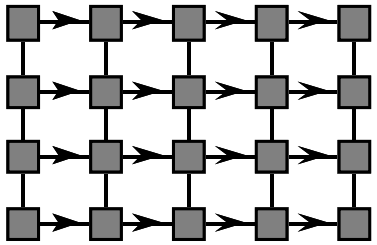
Reactivating **all** PEs inside a selection or loop (possible nested)

```
IF x>0 THEN ... (* only grid PEs are active, which satisfy condition *)
            ALL grid DO
              ... (* all grid PEs are active, regardless of condition *)
            END;
       ELSE ... (* only grid PEs are active, which do not satisfy condition *)
END;
```

Data Exchange

Structured Data Exchange

- all PEs or a group of PEs participates in a data exchange (not just a pair of PEs)
- neighborhood has been defined via connection declarations



```
y := MOVE.east(x);
```

```
SEND.east(4*x, y);
```
only the sender has to be active

```
y := RECEIVE.north(x);
```
only the receiver has to be active

Data Exchange Functions

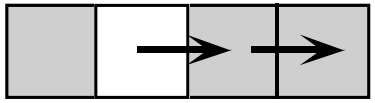
```
CONFIGURATION list[1..max];
CONNECTION right: list[i] -> list[i+1];
VAR x,y: list OF INTEGER;
```

a) Only the sender has to be active.
All active PEs, which have a successor, send data.
`SEND.right(x,y);` Example:

before
x: 1 2 3 4
y: 0 0 0 0

after
x: 1 2 3 4
y: 0 1 0 3

b) Only the receiver has to be active.
All active PEs, which have a predecessor, receive data.
`y := RECEIVE.right(x);` Example:



before
x: 1 2 3 4
y: 0 0 0 0

after
x: 1 2 3 4
y: 1 0 2 3

c) Both sender and receiver have to be active.
All active PEs, which have an active successor, send data.
`y := MOVE.right(x);` Example:

before
x: 1 2 3 4
y: 0 0 0 0

after
x: 1 2 3 4
y: 1 0 3 3

d) Neither sender nor receiver have to be active.
All PEs, which have a successor, send data – independent of their activation status.
This version does not seem to make much sense, so it is not included in Parallaxis (use `ALL` statement instead).

Please note:

- `SEND` is a procedure, `MOVE` and `RECEIVE` are functions
- `RECEIVE` also moves data in the direction specified (not from)
- Data exchange at `CONFIGURATION` boundaries do not cause undefined data

Structured Data Exchange

Modifiers:

```
VAR x,y: list OF INTEGER;
...
SEND.right:2 (x,y);
```

multiple steps — *move data two steps to the east*

Example: *(only PEs 1 and 2 are active, PEs 3 and 4 are inactive)*

	before	*copy*	*1st step*	*2nd step*	*assignment*
x:	**1 2** 3 4	–	–	–	1 2 3 4
intermed. var.:	–	**1 2** 3 4	1 **1 2** 4	1 1 **1 2**	–
y:	0 0 0 0	–	–	–	0 0 1 2

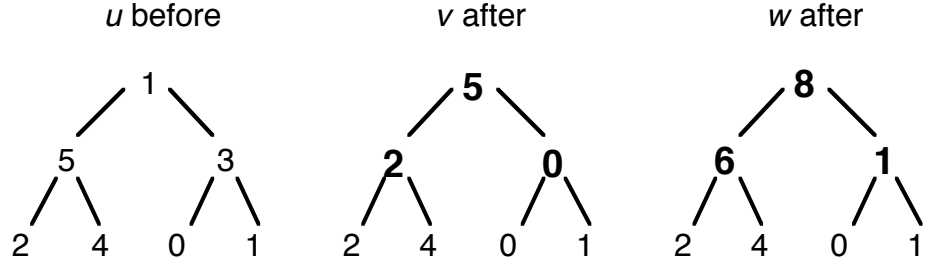
```
VAR u,v,w: tree OF INTEGER;
...
v := u;
IF EVEN(ID(tree)) THEN
  SEND.parent (u,v)
END;#
w := MOVE.parent:#SUM (u);
```

init all components of *v*

moves only the left children's data up the tree

move *and* reduce data

moves data from both children to the parent, resolving multiple arriving data by adding



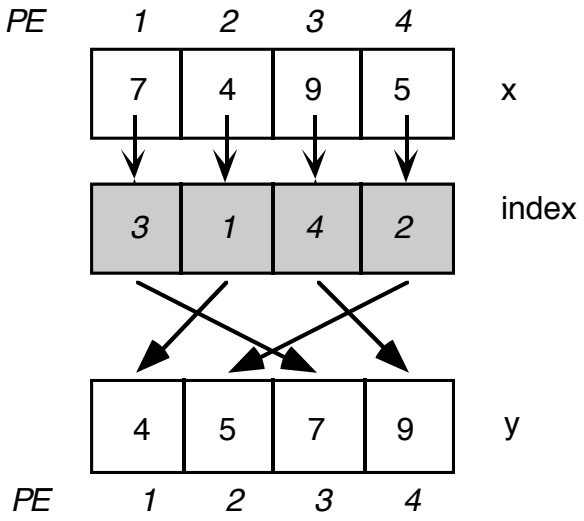
Unstructured Data Exchange

```
VAR x,y,index: grid OF INTEGER;
...
```

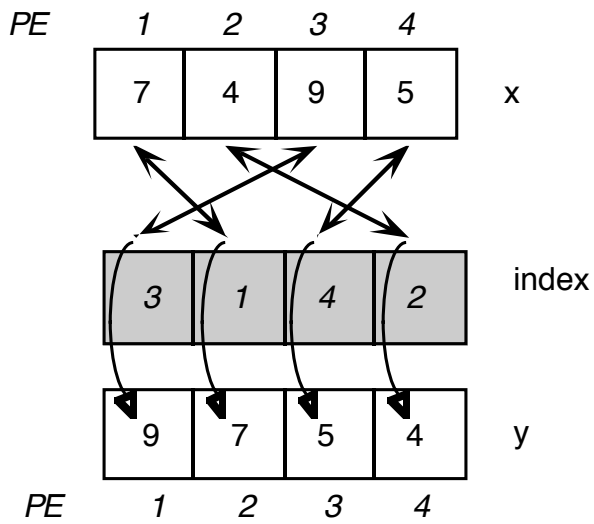
`SEND.<<index>>(x,y);` sends data from all components of `x` to a destination, determined by vector `index`

`y := RECEIVE.<<index>>(x);` receives data from all components of `x` to a destination, determined by vector `index`, however, on the receiver's side

`SEND.<<index>> (x,y) ;`



`y := RECEIVE.<<index>> (x) ;`



Please note: Unstructured data exchange may be an expensive operation
(hardware-dependent, often: everything besides grid is expensive, e.g. MasPar: factor 100)

Unstructured Data Exchange

```
CONFIGURATION grid [1..2],[1..3];
VAR x,y, index,d1,d2: grid OF INTEGER;
...
```

`SEND.<:d2,d1:> (x,y) ;` sends data from all components of `x` to a destination, determined in the first dimension by `d1`, and in the second dimension by `d2`

E.g. $x = \begin{pmatrix} 7 & 3 & 5 \\ 4 & 2 & 3 \end{pmatrix}$ $d2 = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \end{pmatrix}$ $d1 = \begin{pmatrix} 3 & 2 & 1 \\ 3 & 2 & 1 \end{pmatrix}$ **then** $y = \begin{pmatrix} 5 & 2 & 7 \\ 3 & 3 & 4 \end{pmatrix}$

`SEND.<<index>>:#SUM (x,y) ;` data exchange with reduction, if `index` does not provide a 1:1 permutation; positions not indexed get the sender's original elements

E.g. $x = \begin{pmatrix} 7 & 3 & 5 \\ 4 & 2 & 3 \end{pmatrix}$ $index = \begin{pmatrix} 1 & 3 & 2 \\ \mathbf{6} & \mathbf{6} & \mathbf{6} \end{pmatrix}$ **then** $y = \begin{pmatrix} 7 & 5 & 3 \\ \mathit{4} & \mathit{2} & \mathbf{9} \end{pmatrix}$

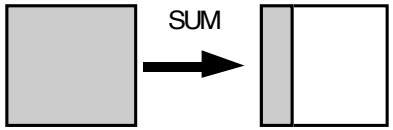
Abbreviations:

```
SEND.<:DIM(grid,2),d1:> (x,y) ;
```
is equivalent to:

```
SEND.<:*,d1:> (x,y) ;
```
use asterisk for dimensions not to be changed

```
SEND.<:d2,LOWER(grid,1):> :#SUM (x,y);
```
is equivalent to:

```
SEND.<:d2,#SUM:> (x,y);
```
collapse dimension 1, accumulate data in the first position of this dimension and permute data in dimension 2 according to `d2`



E.g. $x = \begin{pmatrix} 7 & 3 & 5 \\ 4 & 2 & 3 \end{pmatrix}$ $d2 = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$ **then** $y = \begin{pmatrix} \mathbf{15} & 3 & 5 \\ \mathbf{9} & 2 & 3 \end{pmatrix}$

Selecting Individual PEs

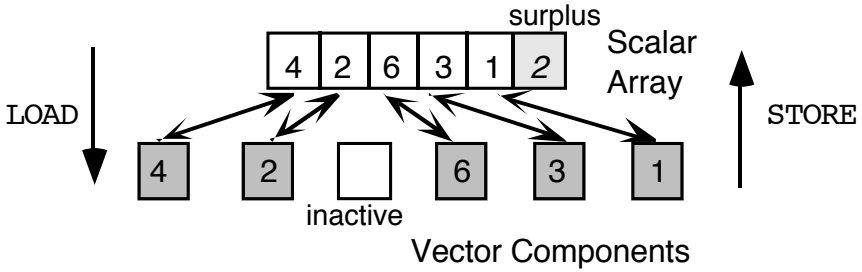
```
VAR x  : grid OF INTEGER;  (* 2-dim. *)
    s,t: INTEGER;          (* scalar *)

...

s := x <<101>> ;           get the value of the PE with ID 101
x <<101>> := s ;           set the value of the PE with ID 101


s := x <:5,t+1:> ;         get the value of the PE in row 5 and column t+1,
                           according to the CONNECTION ranges specified
x <:t,11:> := s ;          set the value of the PE in row t and column 11
```

Exchange between Scalar and Vector Data

```
CONFIGURATION list[1..n];
VAR  s: ARRAY[1..n] OF INTEGER;
     t: INTEGER;
     v: list OF INTEGER;
```

a) Distribute a scalar array componentwise to a vector (`LOAD`) or back (`STORE`).

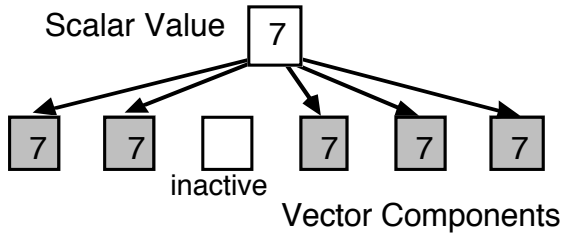
`LOAD (v, s);` from scalar to vector
`STORE(v, s);` from vector to scalar



Time complexity O(*n*) for *n* PEs

b) Assign a scalar to a vector (implicit broadcast)

```
v := t;
```

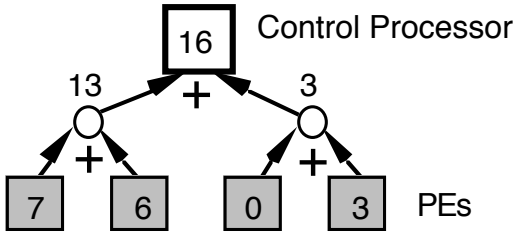


Time complexity O(1) for *n* PEs

Reduction

Operation `REDUCE`: vector → scalar
transforming a vector to a scalar with system-defined operators or any user-defined function.

```
SUM, PRODUCT, MAX, MIN, AND, OR, FIRST, LAST
```




```
VAR s: INTEGER;
    x: grid OF INTEGER;

s := REDUCE.SUM(x);
```

Time complexity $O(\log_2 n)$ for n PEs

User-defined `REDUCE` operation should be associative and commutative to avoid unpredictable results

$$(1 - 2) - 3 \neq 1 - (2 - 3)$$
$$-4 \neq +2$$

Example for user-defined `REDUCE` operation

```
VAR v: grid OF BOOLEAN;
    s: BOOLEAN;
...
PROCEDURE xor (a,b: VECTOR OF BOOLEAN): VECTOR OF BOOLEAN;
BEGIN
  RETURN(a <> b);
END xor;
...
s := REDUCE.xor(v);
```

Modules

- main module
- library modules
  - definition module
  - implementation module
- in Parallaxis also: foreign module (e.g. for linking C routines)

```
DEFINITION MODULE sample;
  EXPORT myproc;
  PROCEDURE myproc(VAR i: INTEGER);
END sample.
```

```
IMPLEMENTATION MODULE sample;
  PROCEDURE myproc(VAR i: INTEGER);
  BEGIN
    i := 2*i + 1
  END myproc;
END sample.
```

```
MODULE mymain;
FROM sample IMPORT myproc;
VAR k: INTEGER;
BEGIN
  k:=0;
  myproc(k);
END mymain.
```

Export/Import

```
FROM sample IMPORT myproc;
...
myproc(k);
```

```
IMPORT sample;
...
sample.myproc(k);
```

Input/Output

`WriteLn;`	Start a new line
`Write(c);`	Write character *c*
`WriteString(s)`	Write string *s*
`WriteInt(i,l);`	Write integer *i* using *l* print spaces
`WriteCard(c,l);`	Write cardinal *c* using *l* print spaces
`WriteReal(r,l);`	Write real *r* using *l* print spaces
`WriteFixPt(r,l,m);`	Write real *r* using *l* print spaces and *m* decimals
`WriteBool(b);`	Write boolean *b*
`Read(c);`	Read character *c*
`ReadString(s)`	Read string *s*
`ReadInt(i);`	Read integer *i*
`ReadCard(i);`	Read cardinal *c*
`ReadReal(r);`	Read real *r*
`ReadBool(b);`	Read boolean *b*
`OpenOutput(s);`	Open file with name *s* for writing (following write operations write to file)
`CloseOutput;`	Close file, redirect output to *stdout* (screen)
`OpenInput(s);`	Open file with name *s* for reading (following read operations read from file)
`CloseInput;`	Close file, redirect input back to *stdin* (keyboard)

Input/Output

```
VAR i: INTEGER;
    r: REAL;
...
WriteString("Hello");     (* write string on screen *)
WriteInt(i,7);            (* write integer value using 7 print spaces *)
WriteLn;                  (* start writing in a new line *)

OpenOutput("myfile");     (* open file, redirect output to file *)
  WriteString("Hello");   (* write string to file *)
  WriteFixPt(r,9,2);      (* write r to file in 9 print spaces with 2 decimals*)
CloseOutput;              (* close file, output back to screen *)

WriteString("Hello");     (* write string on screen *)
```

Sequential Control Structures

- like in Modula-2
- may be used with scalar or vector arguments

```
IF x=0 THEN y:=1; z:=5
       ELSE y:=2
END;
```

```
FOR x:=1 TO 10 DO
  y:=2*y
END;
```

```
WHILE x>0 DO
  x:=x-1; y:=2*y
END;
```

```
REPEAT
  x:=x DIV 2
UNTIL x<7;
```

```
LOOP x:=x-1;
  IF x<7 THEN EXIT
  END;
END;
```

```
CASE x OF
  1,3,7: z:=5; y:=3  |
  8..15: z:=1        |
 ELSE    z:=3; y:=4
END;
```

```
PROCEDURE abc(VAR c: CHAR);
(* param. is "call by reference" *)
BEGIN (* procedure *)
  ...
END abc;
```

```
PROCEDURE def(x: INTEGER): INTEGER;
(* parameter is "call by value" + return value *)
BEGIN (* function *)
  RETURN(x+1)
END def;
```

Difference to Modula-2

- no nesting of local modules
- multiple comparisons

```
IF 7 < x < 12 THEN ... END;
```

- power operator `**`
- constant records and arrays

```
TYPE colorR = RECORD red, green, blue: INTEGER;
              END;
     colorA = ARRAY [1..3] OF INTEGER;
VAR  c: colorR;
     d: colorA;
...
c := colorR(255,255,0);
d := colorA(0,255,255);
```

- access to command line parameters (`argc` and `argv`, similar to C)

```
VAR i  : INTEGER;
    buf: ARRAY [1..20] OF CHAR;
...
FOR i:=1 TO argc DO
  argv(i,buf);
  WriteString(buf); WriteLn;
END;
```

- math and I/O operations are pre-defined and do *not* have to be imported;
  this is necessary, for these procedures and functions can take scalar *or* vector parameters

- `MOD` returns always positive results in Parallaxis
  `(-1) MOD 5 = 4` this is different from C and some Modula-2 implementations: `-1`
  `1 MOD (-5) = 4`

  useful for defining torus connections:

```
CONFIGURATION ring[0..max-1];
CONNECTION    left: ring[i] -> ring[(i-1) MOD max];
```

Parallel Concepts Reviewed

`CONFIGURATION`	←	specify PE arrangement (similar to array declaration)
`CONNECTION`	←	specify connections between PEs
config_name `OF` *type*	←	vector data type
`VECTOR`	←	generic vector parameter in procedures
`IF` *vector_condition* `THEN`	←	implicit PE selection, parallel `IF`
`WHILE` *vector_cond* `DO`	←	implicit PE selection, parallel `WHILE` loop
`ALL`	←	reactivate all PEs in a selection
`ID / DIM`	←	position values for PEs
`MOVE / SEND / RECEIVE`	←	data exchange
`REDUCE`	←	reduce a vector to a scalar
`LOAD`	←	send scalar array to PEs
`STORE`	←	send vector data to scalar array

Data Parallel Algorithms

- programming of SIMD systems
  (single instruction stream, multiple data stream)
- completely different approach from MIMD algorithms
  (multiple instruction stream, multiple data stream)
- efficiency is important, but PE utilization is not the outmost goal
  (surplus PEs cannot be used for other tasks)
- natural design of algorithms
- simple, easy to understand programs

Sample Algorithms in Parallaxis

- edge detection in images
- cellular automata
- fractals
- sorting
- stereo image analysis

Edge Detection in Images

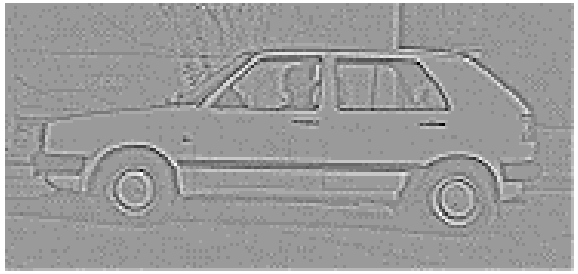
Laplace Operator

For each pixel:

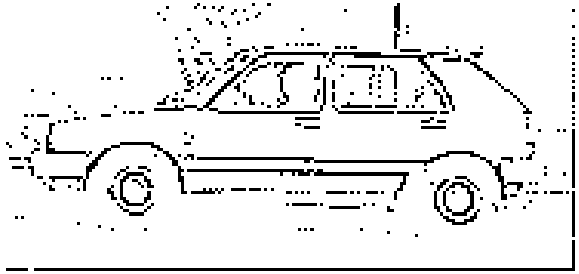
	–1	
–1	4	–1
	–1	

In the whole image:

Original (only intensity is used)



Filtered with Laplace



Threshold to generate binary image

```
PROCEDURE Laplace(x: grid OF INTEGER): grid OF INTEGER;
BEGIN
  RETURN( 4*x - MOVE.north(x) - MOVE.south(x) - MOVE.east(x) - MOVE.west(x) );
END Laplace;
```

Sobel Operator

Two operators for each pixel:

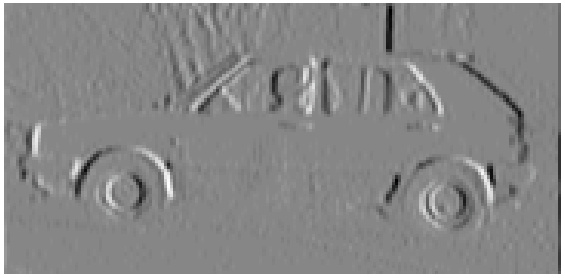
–1		1
–2		2
–1		1

1	2	1
–1	–2	–1

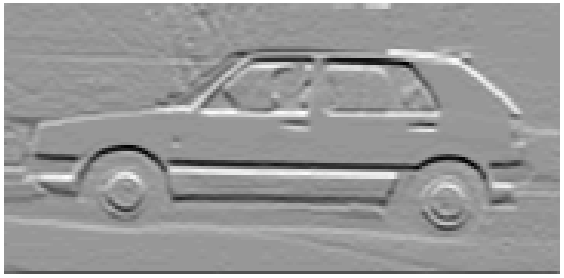
In the whole image:

Intermediate steps

vertical edges only

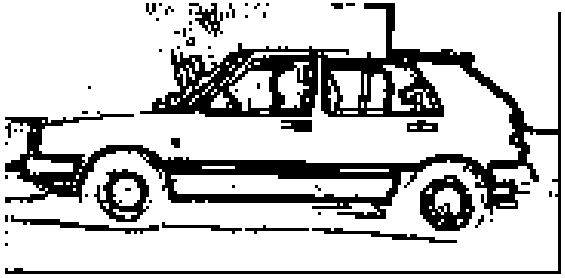


horizontal edges only



Final result

edges = $\sqrt{\text{vertical}^2 + \text{horizontal}^2}$



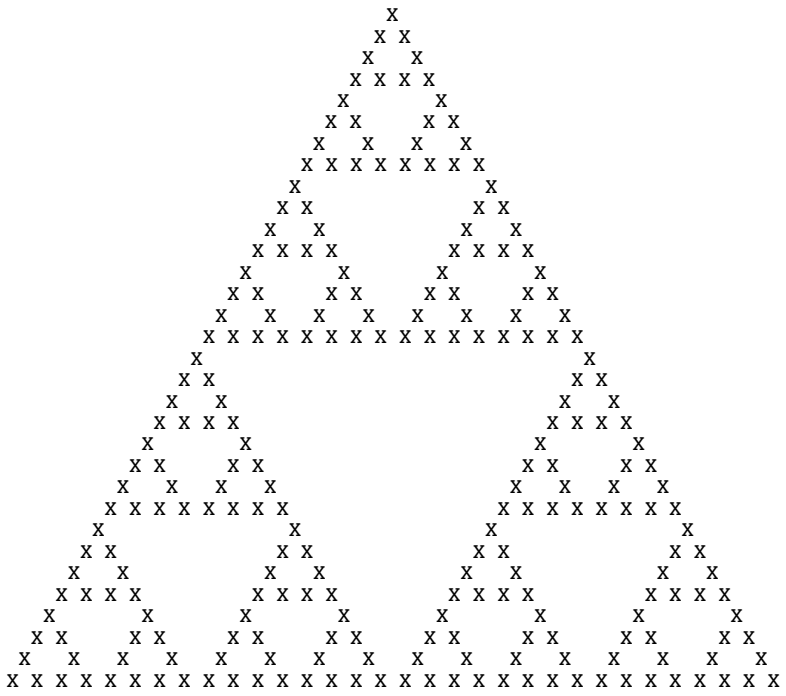
```
TYPE i_image = grid OF INTEGER;

PROCEDURE sobel_x (a: i_image): i_image;
...
END sobel_x;
```

```
PROCEDURE sobel_y (a: i_image): i_image;
...
END sobel_y;
```

```
PROCEDURE Sobel(a: i_image): i_image;
BEGIN
  RETURN( TRUNC(sqrt(sobel_x(a) ** 2 + sobel_y(a) ** 2)) );
END Sobel;
```

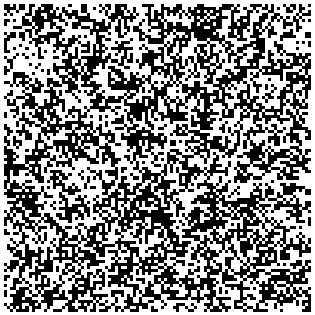
Cellular Automata

```
MODULE cellular_automaton;
CONST n = 79;           (* num. elements *)
      m = (n+1) DIV 2;  (* num. loops    *)
CONFIGURATION list [1..n];
CONNECTION left: list[i] <-> list[i-1] :right;

VAR i  : INTEGER;
    val: list OF BOOLEAN;
    c  : list OF ARRAY BOOLEAN OF CHAR;

BEGIN
  val     := ID(list) = m; (* Init *)
  c[FALSE]:= " ";
  c[TRUE] := "X";
  FOR i := 1 TO m DO
    Write(c[val]);;
    val := MOVE.left(val) <> MOVE.right(val);
  END;
END cellular_automaton.
```

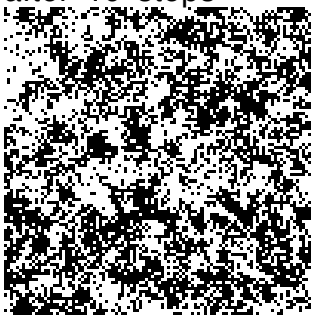


More Cellular Automata

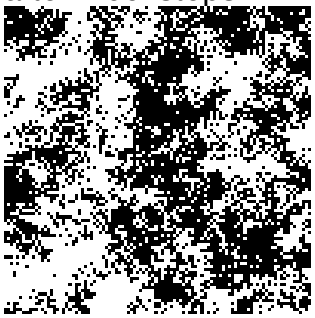
Random Init



after 10 steps



after 100 steps



Simulating the voting behaviour of a population

- random initialization
- for each step: each PE takes the opinion of a randomly selected neighbor
- a cluster results

```
PROCEDURE vote;
VAR step   : INTEGER;
    opinion: grid OF BOOLEAN;
    n      : grid OF ARRAY[0..7] OF BOOLEAN;  (* neighbors *)
BEGIN
  opinion := RandomBool(grid);  (* init *)
  FOR step:=1 TO max_steps DO
    get_neighbors(opinion,n);
    opinion := n[ RandomInt(grid) MOD 8 ];
    ... (* write current state as image *);
  END;
END vote;
```

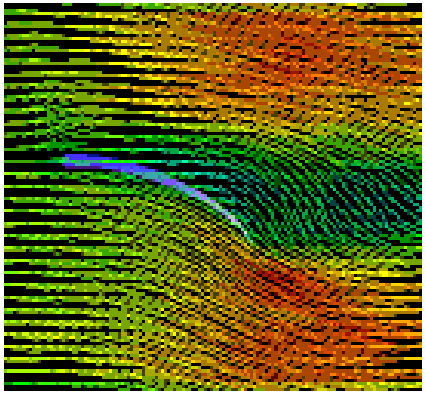
(user function `get_neighbors` returns state values of all 8 neighbors in an array)

(see also Quicktime animation)

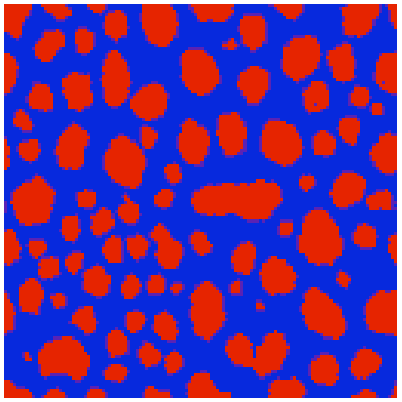
Lattice Gas Automata

- simulation of gas and fluid dynamics
- discretization of space and time
- hexagonal grid for simulation
- only local data exchange required

Flow field on a wing profile

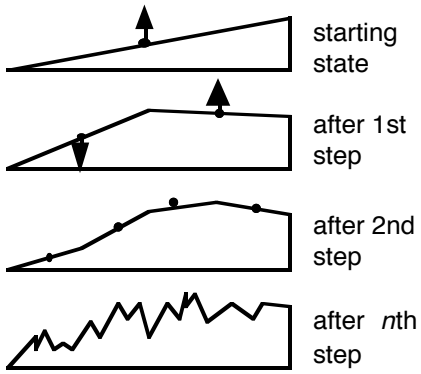


Immiscible fluids (e.g. oil in water)

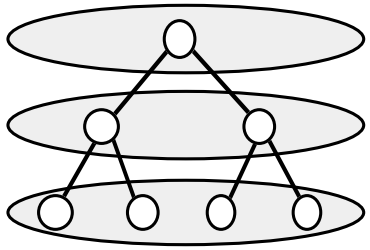


(see also Quicktime animation)

Fractals



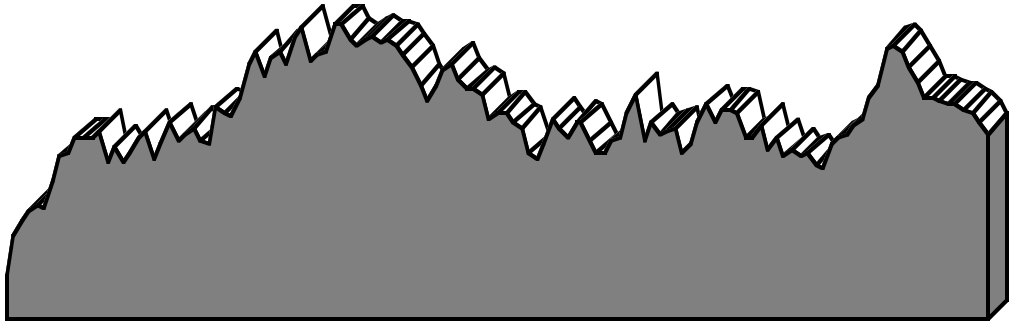
PEs Arranged as a Binary Tree



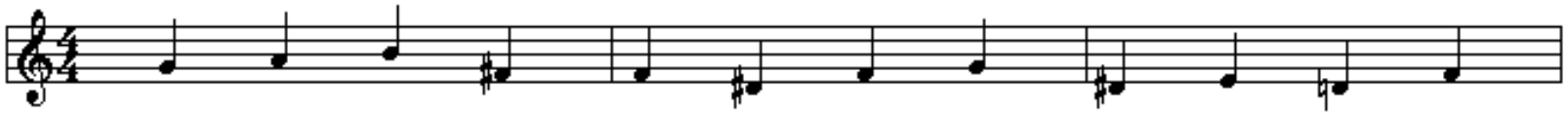
```
CONFIGURATION tree [1..maxnode];
CONNECTION    lchild : tree[i] <-> tree[2*i]   :parent;
              rchild : tree[i] <-> tree[2*i+1] :parent;

VAR low, high, x: tree OF REAL;

PROCEDURE MidPoint(delta: REAL; level: INTEGER);
BEGIN   (* select level *)
  IF 2**(level-1) <= ID(tree) <= 2**level-1 THEN
    x := 0.5 * (low + high) + delta*Gauss();
    IF level < maxlevel THEN
      SEND.lchild (low,low);  (* values for children *)
      SEND.lchild (x,high);
      SEND.rchild (x,low);
      SEND.rchild (high,high);
    END;
  END;
END MidPoint;
```

Computed Fractal Curves

Interpreted as a mountain range

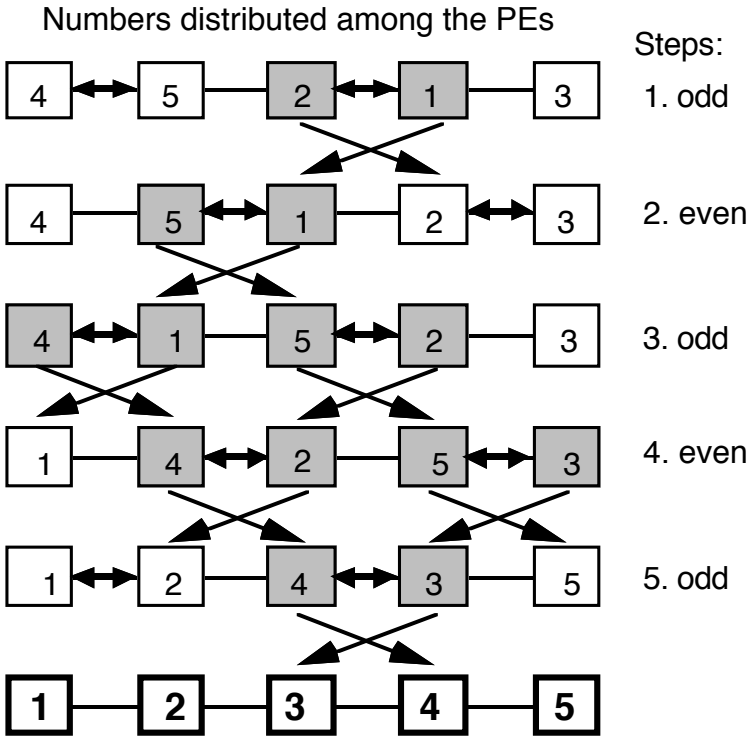


Interpreted as a musical score



Sorting

Odd-Even Transposition Sorting
(a kind of parallel Bubblesort)

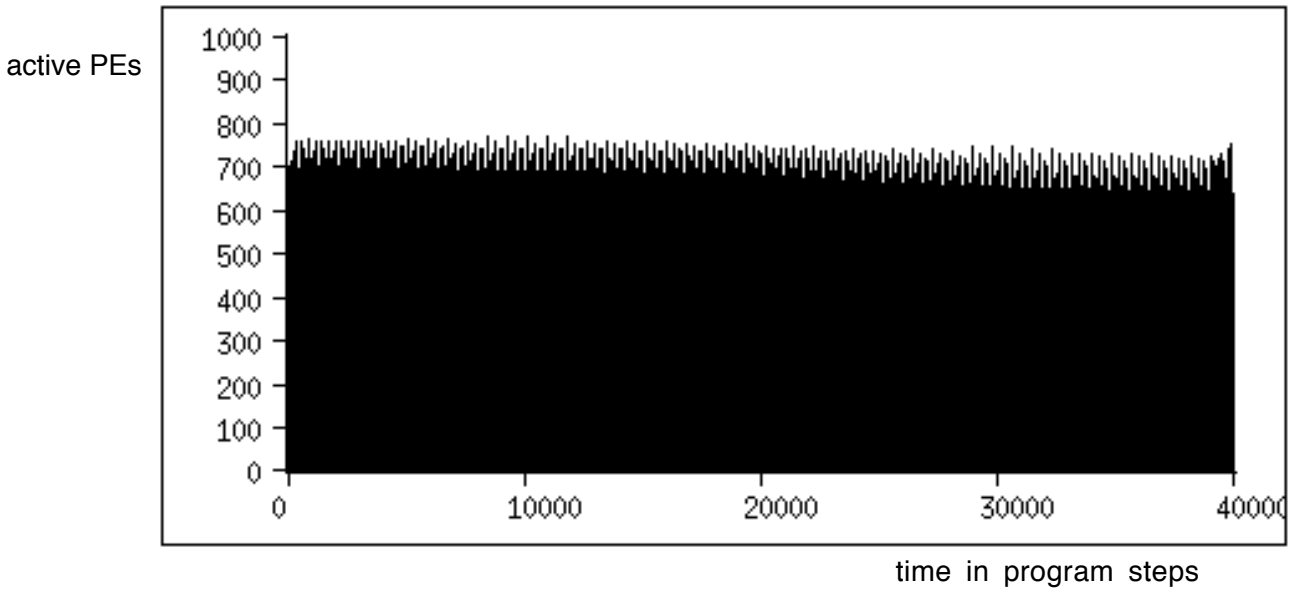



```
MODULE sort;  (* Odd-Even Transposition Sorting *)
CONST n = 10;
CONFIGURATION list [1..n];
CONNECTION left : list[i] <-> list [i-1] :right;
VAR step    : INTEGER;
    a       : ARRAY[1..n] OF INTEGER;
    val,comp: list OF INTEGER;
    lhs     : list OF BOOLEAN;

BEGIN
  WriteString('Enter 10 values: ');
  ReadInt(val);
  lhs := ODD(ID(list));  (* left-hand-side of a comp. *)
  FOR step:=1 TO n DO
    IF lhs THEN comp := RECEIVE.left(val)
           ELSE comp := RECEIVE.right(val)
    END;
    IF lhs = (comp<val) THEN val:=comp END;
    lhs := NOT lhs;     (*    lhs & (comp< val) *)
  END;                  (* or rhs & (comp>=val) *)
  WriteInt(val,5);
END sort.
```

(see also Quicktime animation)

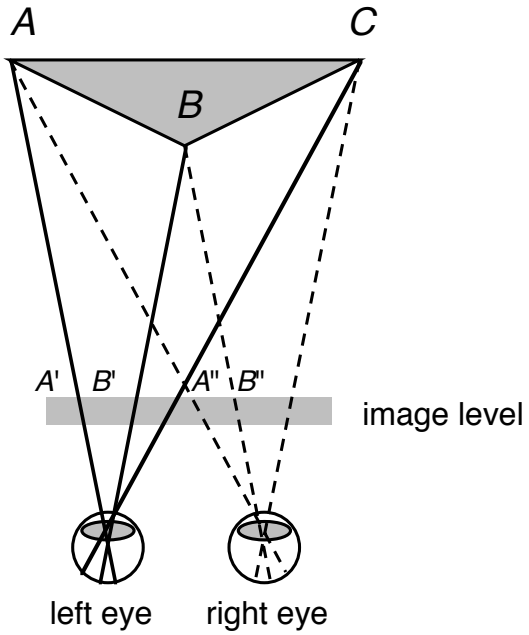
Sorting 1000 Numbers

(processor utilization without I/O operations)



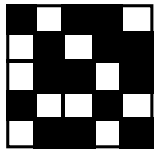
Stereo Image Analysis

- easy task for people, **but**
- extremely compute-intense problem
- may be solved data-parallel with local operations

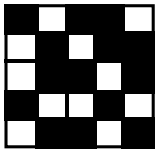


Generation of Random Dot Stereograms

1. Fill the left and right images with identical random values

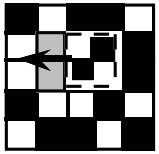


left

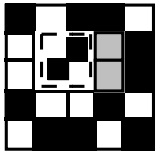


right

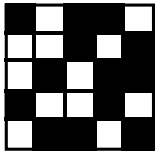
2. Raising or lowering areas



right image before



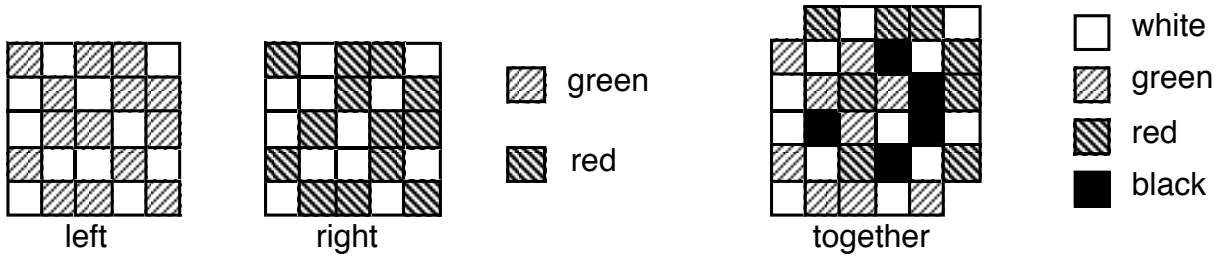
right image after

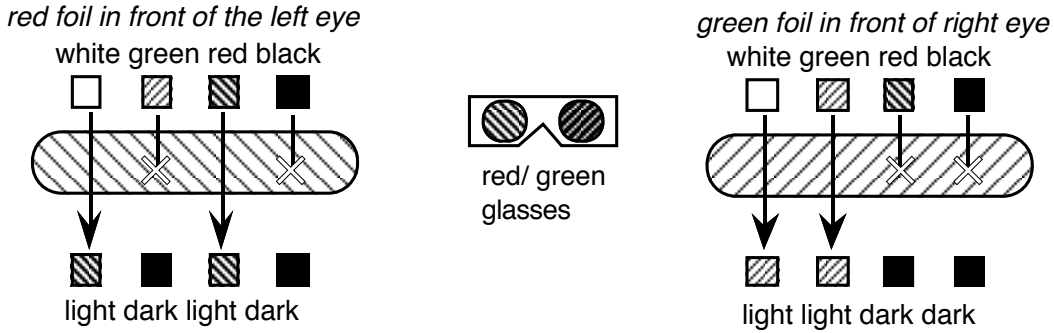


right image filled

```
TYPE b_image = grid OF BOOLEAN;
     g_image = grid OF [0..255];
     i_image = grid OF INTEGER;
...
PROCEDURE generate_random_dot(VAR l_pic, r_pic: b_image);
VAR i,num, from_x, from_y, to_x, to_y, shifts: INTEGER;
BEGIN
  l_pic := RandomBool(grid);  (* random vector boolean *)
  r_pic := l_pic;
  WriteString("Number of Areas to Elevate: ");
  ReadInt(num);
  FOR i := 1 TO num DO
    WriteString("Area: from-x from-y to-x to-y shifts ");
    ReadInt(from_x); ReadInt(from_y);
    ReadInt(to_x); ReadInt(to_y);
    ReadInt(shifts);
    IF (from_y <= DIM(image,2) <= to_y) AND
       (from_x - shifts <= DIM(image,1) <= to_x) THEN
      SEND.left:shifts (r_pic,r_pic) (* move rectangle *)
    END;
    IF (from_y <= DIM(image,2) <= to_y) AND
       (to_x - shifts <= DIM(image,1) <= to_x) THEN
      r_pic := RandomBool(grid);  (* fill gap *)
    END;
  END;
END generate_random_dot;
```

Display of Stereograms

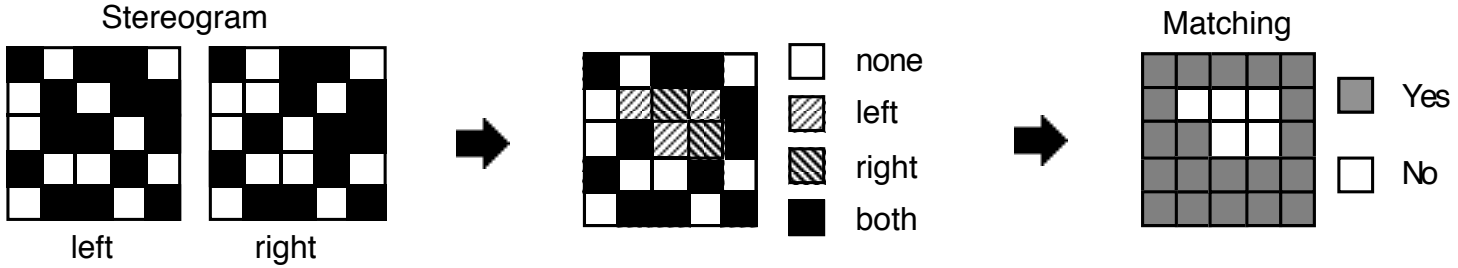




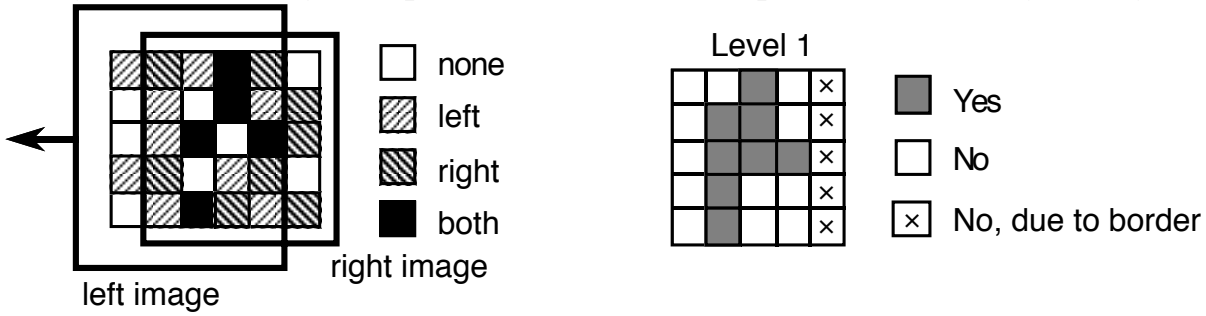
Analysis of Random Dot Stereograms

(Reverse calculation of depth information from stereograms)

1. Comparison of the right image to the left image and search for matching pixels

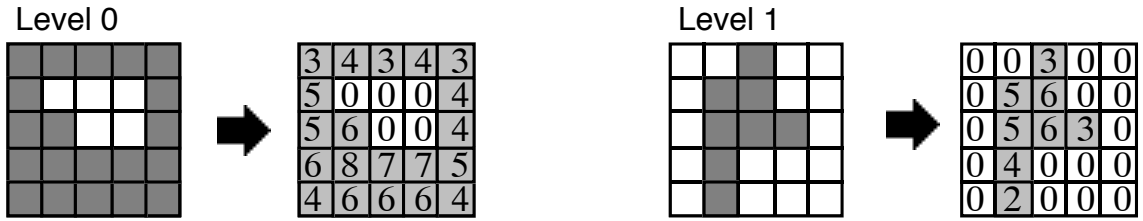


2. Shifting of the left image one pixel to the left and comparison with the right image



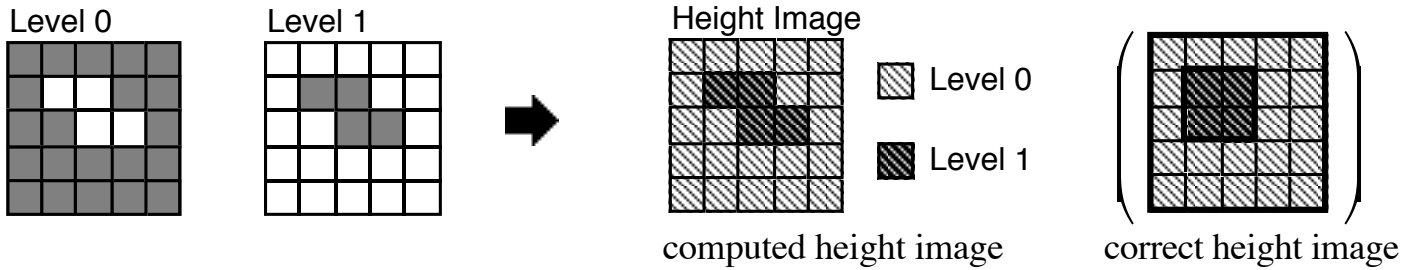
Iteratively for each depth level

3. Determining the depth of each Pixel



Iteratively for each depth level

4. Selection of the best fitting level (depth) for each pixel



5. Filter (optional)
Reduce image defects (noise)

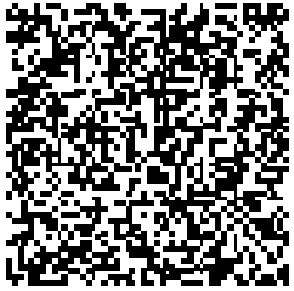
Stereogram Analysis Program

```
PROCEDURE analyze_random_dot(l_pic,r_pic: b_image;
                             steps: CARDINAL; VAR elev: g_image);
VAR equal          : b_image;
    level, maxlevel: i_image;
    i              : INTEGER;

  PROCEDURE sum_3x3(val: i_image): i_image;
  VAR l_r: i_image;
  BEGIN
    l_r :=  val + MOVE.left(val) + MOVE.right(val);   (* horizontal sum          *)
    RETURN( l_r + MOVE.up(l_r)   + MOVE.down(l_r) );  (* cumulative vertical sum *)
  END sum_3x3;

BEGIN
  elev     := 0;
  maxlevel := 0;
  FOR i := 0 TO steps DO (* add congruences in 5x5 neighborhood *)
    equal := l_pic = r_pic;
    level := sum_3x3( ORD(equal) );
    (* find image plane with max. value *)
    IF equal AND (level > maxlevel) THEN
      elev     := i;
      maxlevel := level;
    END;
    l_pic := MOVE.left(l_pic);  (* move image *)
  END;
END analyze_random_dot;
```

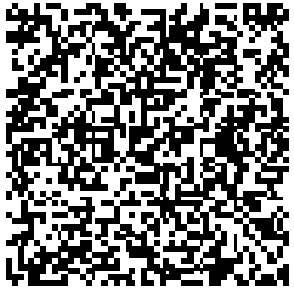
Program Results and PE Utilization

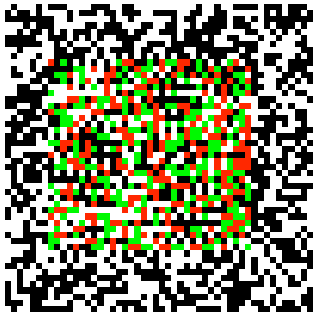
Generated Random Dot Stereogram:

left image

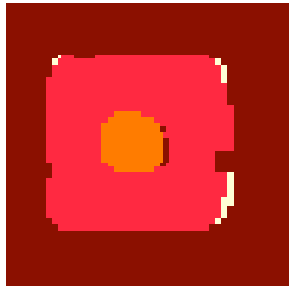


right image





red/green stereogram



computed depth image

PE utilization for 100 × 100 Pixels: