# Parallaxis-III: A Language for Structured Data-Parallel Programming

Thomas Bräunl
Computer Science Dept., IPVR
Computer Vision Group
Univ. Stuttgart, Germany

**Abstract:**
Parallaxis is a machine-independent language for data-parallel programming. Sequential Modula-2 is used as the base language. Programming in Parallaxis is done on a level of abstraction with virtual processors and virtual connections, which may be defined by the application programmer. This paper describes Parallaxis-III, the current version of the Parallaxis language together with a number of parallel sample algorithms.

# 1. Parallel Language Constructs

*Parallaxis* [Bräunl 89], [Bräunl 91], [Bräunl 93] is based on Modula-2 [Wirth 83], extended by data-parallel concepts. The language is fully machine-independent across SIMD architectures; therefore programs written in Parallaxis run on different SIMD parallel computer systems. For a large number of (single-processor) workstations and personal computers, there is also a Parallaxis simulation system with source level debugging and tools for visualization and timing. A compiler for data-parallel programming of MIMD computer systems in SPMD style (same program, multiple data) is being developed. Parallel programs with small data sets can be developed, tested and debugged with this simulation system. Then, Parallaxis compilers can be used to generate parallel code for the MasPar MP-1 / MP-2 or the Connection Machine CM-2. The simulation environment allows both the study of data parallel fundamentals on simple computer systems and the development of parallel programs which can later be executed on expensive parallel computer systems. The programming environment for Parallaxis is available as public domain software [Barth, Bräunl, Engelhardt, Sembach 92].

We had two major goals in developing a new parallel programming language:

1. We believe that 'structured programming' has a number of advantages in developing sequential *and* parallel software, as well as for learning programming concepts in general. Therefore, we chose Modula-2 as base language and not C like many other approaches.

2. Almost all commercial SIMD programming languages are machine-dependent, since they have been specifically designed for a single hardware platform to achieve maximum performance. Therefore, one may not easily port a program from one SIMD system to another. To avoid this problem, we designed Parallaxis to be completely machine-independent. This refers to SIMD architectures only, for switching from SIMD to MIMD will almost always require to change the *algorithm* of your program, in order to keep it efficient.

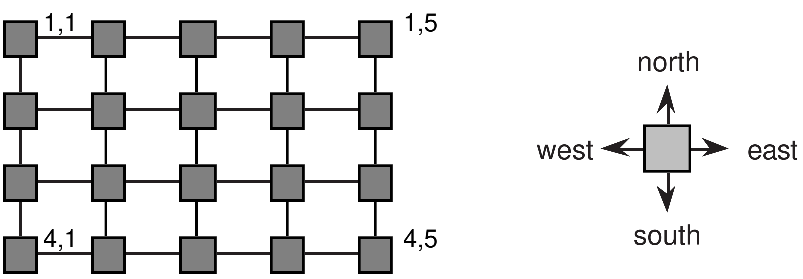
The central point of Parallaxis is programming on a level of abstraction with virtual PEs and virtual connections. In addition to the algorithmic description, every program includes a semi-dynamic connection declaration in functional form. This means that the desired PE network topology is specified in advance for each program (or for each procedure) and can be addressed in the algorithmic section with symbolic names (instead of complicated arithmetic index or pointer expressions). However, for completeness, also full-dynamic data exchange operations may be performed.

The following describes the latest version of the language, *Parallaxis-III*, which is not fully compatible to older versions of the language.

## 1.1. Specifying Virtual Processors and Connections

One or several 'virtual machines' consisting of processors and a connection network may be defined for every Parallaxis program. This is done in two simple steps. First, the keyword `CONFIGURATION` is used to specify the number of PEs and their arrangement in analogy to an array declaration. However, at this point, no specification has been made as to the connection structure between the PEs. This follows by specifying mapping relations, introduced by the keyword `CONNECTION` (this second step may be omitted if no connections are required). Every connection has a symbolic name and defines a mapping from a PE (*any* PE) to the corresponding neighbor PE. The specification of this *relative* neighbor is accomplished by providing an arithmetic expression for the index of the destination PE. Data exchanges can now be carried out using these symbolic connection names in the parallel program.

Figure 1.1 shows a PE arrangement as a two-dimensional grid structure in a simple Parallaxis example. The `CONFIGURATION` declaration provides $4 \times 5$ virtual processors, which are virtually connected to one another in the following `CONNECTION` declaration In the figure, PE 1,1 is in the upper left corner; however, you may also choose to put the 'origin' in the lower left corner – it does not matter as long as you use it consistently in your program. Since homogeneous connection structures or topologies are easy to declare, four connection declarations are sufficient to construct a grid of any size. One connection is defined for each cardinal direction. The connection to the north, for example, decrements the first index. Some connections from the border PEs lead 'nowhere', which means that these connections do not exist, and will not participate in any data exchange operation.



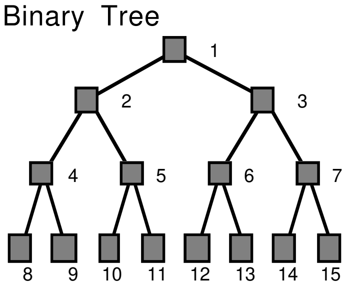

```
CONFIGURATION grid [1..4],[1..5];
CONNECTION  north: grid[i,j] →  grid[i-1, j];
            south: grid[i,j] →  grid[i+1, j];
            east : grid[i,j] →  grid[i, j+1];
            west : grid[i,j] →  grid[i, j-1];
```

**Figure 1.1** Two-dimensional grid topology with representative PE

In case one prefers a wrapped-around torus instead of an open grid for some application, this can be easily accomplished by using the modulo-operator (remainder). Identifiers `h` and `w` denote constants:

```
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];
```

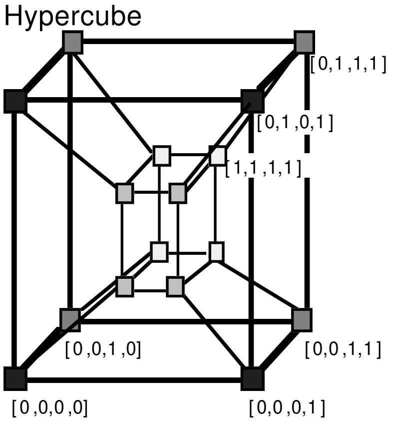
There is a list of extensions to this simple process of defining virtual machine structures: Several destination expressions may be specified after the arrow symbol, separated by commata. Connections may be *parameterized*, as with the hypercube in Figure 1.2. With these, it is possible to perform a data exchange in a *computed* direction. For the definition of the binary tree network, also in Figure 1.2, *bi-directional* connections ('↔' or '<->' in ASCII notation) have been used instead of *uni-directional* connections ('→' or '->'). A bi-directional connection is an abbreviation for two uni-directional connections and, therefore, requires a second connection name on the right hand side of the connection.




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

an alternative specification without distinguishing between left and right children is:

```
CONFIGURATION tree [1..15];
CONNECTION child: tree[i] → tree[2*i],
                            tree[2*i+1];
```




```
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];
```

**Figure 1.2** Tree and hypercube topology

So-called 'compound connections' may be used to have a case distinction inside a connection. The following example connects local pairs of PEs:

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


**Figure 1.3** Compound connections

A case distinction is made for the `next` connection. If the PE-number is odd, a connection to the right neighbor is established, while if it is even, a connection to the left neighbor is established. Using compound connections, arbitrary connection structures, even with irregularities, may be defined. If several parts of a compound connection evaluate to `TRUE` for a particular PE, then for this PE all of these connections are defined (one-to-many or 1:*n* connection).

The connection structures as defined by the `CONNECTION` declaration do not have to be 1:1 (one-to-one) connections. For 1:*n* connections, an implicit broadcast is executed. In the following example, the first element of each row is connected to all elements in this row.

```
CONFIGURATION grid [1..100],[1..100];
CONNECTION one2many: grid[i,1] →  grid[i,1..100];
```

If a one-to-many connection is to be established to all PEs of a dimension, the range may be substituted by an asterisk '*'. So the following is equivalent to the previous example.

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

However, for *n*:1 (many-to-one) connections (and the general *m*:*n* connections, many-to-many) one must ensure that only a single value arrives at any PE's entry port. Therefore, each data exchange operation may include a vector reduction, as will be discussed later. In the following example, all elements of a row are connected to the first element in their column.

```
CONFIGURATION grid [1..3],[1..5];
CONNECTION many2one: grid[i,j] →  grid[i,1];
```

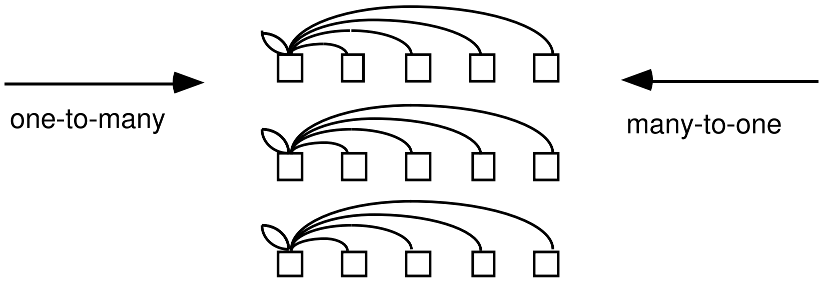



**Figure 1.4** Multiple connections

## 1.2. Multiple Configurations and Iterative Connections

In addition to the constructs shown so far, the definition of multiple topologies in a program is possible. These may be defined independently of each other on separate groups of PEs – in which case the topologies may have different vector data structures. Or the topologies can be defined as 'different views' of the same set of PEs with identical data structure. Furthermore, local topologies may be defined in procedures, thus allowing semi-dynamic connection structures.

Different configuration definitions denote different sets of PEs. For example, the following definition defines two **distinct sets** of PEs:

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

On the other hand, configurations defined within the same declaration structure denote different views of the **same set** of PEs. In this case, the numbers of PEs have to be identical.

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

Connections may be specified between multiple configurations, whether they belong to the same or distinct sets of PEs:

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

However, a connection may only link PEs within the same or two different configurations. A connection between three or more configurations is not allowed, and a connection linking PEs within one configuration *and* PEs to another configuration (by the same connection name) is not allowed.

Configurations only come to life, when they are used for the declaration of a vector variable, which is later used in a computation. A vector variable based on a configuration with multiple views can make use of all connection structures defined for any of the views. Such a variable is also assignment compatible to a variable of the same type but of a different view.

An extension to the definition of connections is the use of iterative connection functions, as exemplified by the following alternative definition for a hypercube network of arbitrary size (the exponentiation function is denoted by '`**`', `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;
```

If *n* equals 10, there are 1,024 PEs defined together with ten bi-directional connections. Expression `EVEN(i DIV 2**k)` tests, whether the `k`-th bit of `i` equals 0.

A large program may be split into several modules, which are compiled separately. So, e.g. for a module containing library functions, it may be desirable *not* to specify the size of a configuration. When writing routines for image processing, the size of the grid structure should be left unspecified and will be defined later by the module importing these routines. An open configuration is indicated by using an asterisk '`*`' instead of a value range. The configuration size may be determined dynamically at run time, e.g. by passing a parameter that is subsequently used as a bound in the configuration declaration.

```
DEFINITION MODULE Open;
CONFIGURATION grid[*],[*];
CONNECTION left: grid[i,j] <-> grid[i  ,j-1] :right;
           up  : grid[i,j] <-> grid[i+1,j  ] :down;
PROCEDURE sum_3x3(input: grid OF INTEGER): grid OF INTEGER;
END Open.
```

```
MODULE Main;
FROM Open IMPORT grid, sum_3x3;
CONFIGURATION my_grid = grid[1..10],[1..10];
VAR a,b: my_grid OF INTEGER;
BEGIN
  a := 1;
  b := sum_3x3(a);
  WriteInt(b,5);
END Main.
```

Open configurations are needed when a procedure is to work on a vector of unspecified size, but has to make use of connections for data exchange or position data (`DIM`, see section on processor positions). If connections and position data are not required in a procedure, which is to be used for different configurations (different size or arbitrary configuration), then the simpler concept of generic vector parameters may be used (`VECTOR`, see section on data declaration).

## 1.3. Data Declaration

Parallaxis differentiates between scalar and vector variables in data declarations as well as in procedure parameters and results. Scalar data is placed on the control processor, while vectors are distributed component-wise among the virtual PEs (see Figure 1.5, here configurations are not connected). Since a program may contain several PE configurations with different numbers of PEs, the name of a configuration is used for specifying the vector type of a variable.

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

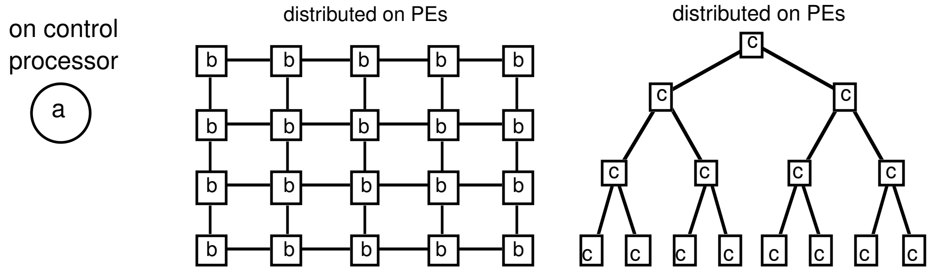



**Figure 1.5** Allocation of scalar and vector data

There is, of course a fundamental difference between declaring an array of vectors and declaring a vector of arrays, as exemplified in Figure 1.6. However, both indexed expressions `x[1]` and `y[1]` refer to a data structure of the same type "`grid OF INTEGER`".

```
VAR x: ARRAY[1..3] OF grid OF INTEGER;
    y: grid OF ARRAY[1..3] OF INTEGER;
```

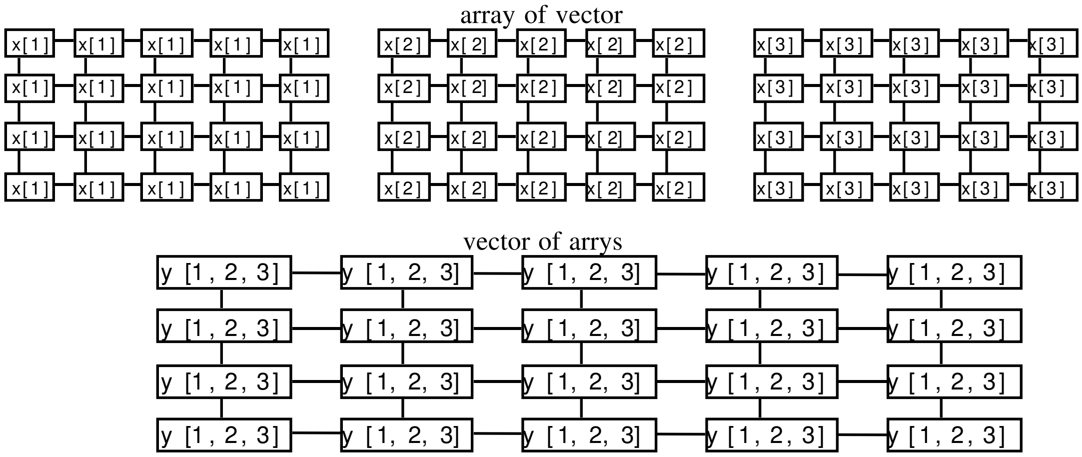



**Figure 1.6** Array of vectors and vector of arrays

Unfortunately, this declaration semantics has some unpleasant effect for procedure arguments. Imagine, e.g. writing a function `factorial`, for computing the factorial value for an argument of type `INTEGER`. Now, a `factorial` function would have to be declared for scalar arguments, and for *every configuration defined* in a program. Since there is no way of knowing them in advance, it would be impossible to write general library routines. To remedy this situation, parameters and variable declarations inside such a procedure may use the keyword `VECTOR` instead of a particular configuration name. This indicates that a parameter will be used in a parallel computation, without specifying a particular configuration (this results in a *generic procedure*). All parameters declared as generic vectors or variables in such a procedure have to belong to the *same* configuration. Since no particular configuration has been specified, no data exchange may be performed in such a procedure.

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

PROCEDURE v_factorial(a: VECTOR OF INTEGER): VECTOR OF INTEGER;
VAR b: VECTOR OF INTEGER;          (* any vector *)
  ...
END v_factorial;
```

It is possible that several procedure parameters use the "open vector" or the "open configuration" construct for **possibly different** configurations. In this case, a local variable declaration with the method used above is not unique. Therefore, the name of the "open vector" or open configuration parameter may follow the keyword `VECTOR` or the open configuration name in the type specification.

```
PROCEDURE vec(a: VECTOR OF INTEGER; b: VECTOR OF INTEGER);
VAR x: VECTOR a OF INTEGER; (* vector type corresponding to a *)
    y: VECTOR b OF INTEGER; (* vector type corresponding to b *)
  ...
END vec;
```

```
CONFIGURATION grid[*],[*];

PROCEDURE open(a: grid OF REAL; b: grid OF REAL);
VAR x: grid b OF REAL;   (* vector type corresponding to b *)
  ...
END open;
```

## 1.4. Processor Positions

There are two ways to determine a PE's current position. The first is by using the (vector-valued) standard function `ID`, which returns the virtual processor position as a single number in row-major ordering (or ‘highest-dimension-major’ for more than two dimensions):

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

1st dimension

2nd dimension

This results in each component of $x$ being assigned the number of its virtual PE, **always** starting with ‘1’ (independent of the configuration range and number of dimensions):

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

The second way of determining the position of a virtual PE, now with respect to its configuration declaration, is to use the standard function `DIM`. This function takes the configuration name and the number of the dimension as arguments and returns the position of a PE within this dimension. Dimensions are numbered from right to left, that is, the highest dimension is at the leftmost position.

The position values returned by function `DIM` depend on the ranges of the configuration definition. The `DIM` function will return exactly this range of values as position data. In the following, position data for rows and columns of the grid example above are shown:

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

Functions, `ID` and `DIM`, as well as the following functions, do not depend on any PE connections. Further functions working on configuration data are shown below. These functions take either configurations or vector variables as arguments.

| | | |
|---|---|---|
| `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) |

A function and a procedure are provided to transfer `ID` values into `DIM` values and vice versa. An array is used for `DIM` values. Both routines may be used with scalar or vector parameters. These routines are especially useful for converting `ID` and `DIM` values of multiple configurations.

```
VAR dims: ARRAY[1..2] OF INTEGER;
    s   : INTEGER;
...
dims[1] := 1;
dims[2] := 2;
s := DIM2ID(grid,dims);
...
ID2DIM(grid,12,dims);
```

| | |
|---|---|
| `s := DIM2ID(grid,dims);` | *s* becomes 9  (referring to grid definition above) |
| `ID2DIM(grid,12,dims);` | *dims*[1] becomes –1<br>*dims*[2] becomes +3 |

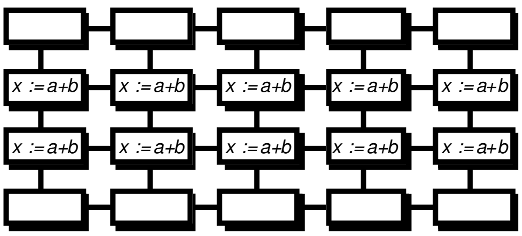
An individual PE may be selected by using an `IF`-selection involving parallel position data like `ID` and `DIM`. However, supplying the position values directly for a vector is also possible and gives a simpler way of selecting PEs. A single PE may be selected using a component expression with scalar arguments after the identifier of a vector variable:

```
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 `CONFIGURATION` ranges specified |
| `x <:t,11:> := s ;` | set the value of the PE in row t and column 11 |

## 1.5. Parallel Execution

Parallel execution is implicit in Parallaxis-III, depending on the declaration of variables involved in a statement or expression. PE-selection (determining which PEs will be active during a certain statement) is also implicit. Any selection or iteration instruction (`IF`, `FOR`, `WHILE`, `REPEAT`, `CASE`, `LOOP`) with a vector argument may be used. Figure 1.7 shows the data-parallel execution of a statement on a selected group of PEs.




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

**Figure 1.7** Data parallel instruction

So whenever a selection is performed, e.g. by an `IF` statement with vector condition, only those PEs are active during execution of the `THEN` branch, whose local condition evaluates to `TRUE`. A `THEN` branch or an `ELSE` branch will only be executed if the condition (or its negation, respectively) will be satisfied by at least one PE. Primarily, this has an effect on any scalar statements that may be in such a branch, since in this case they will be skipped. In the general case, when the condition evaluates to `TRUE` for some PEs, but evaluates to `FALSE` for some other PEs, then both `THEN` branch and `ELSE` branch will be executed subsequently (first `THEN`, afterwards `ELSE`) with the appropriate group of PEs being active (this also holds for any scalar statements that may be contained in these branches). If vector `IF` statements are nested, then in one of the inner branches only a subset of the PEs of the corresponding outer branch is active.

```
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 | |

When entering a loop with vector condition (e.g. `WHILE` loop), only those PEs are active which satisfy the condition. In subsequent iterations of this loop, the number of PEs is *always decreasing*. The loop iterates until no PE is left to satisfy the loop condition (see the following example).

```
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 | |

Other control structures, known from sequential Modula-2 may be used as well in vector context. The `CASE`-selection can be treated as a nested chain of `IF-THEN-ELSIF`-

selections, while FOR- and REPEAT-loops can be regarded as modifications of a WHILE-loop. The parallel LOOP-EXIT construct differs from its sequential counterpart, for it has to specify the name of the selected configuration, e.g.:

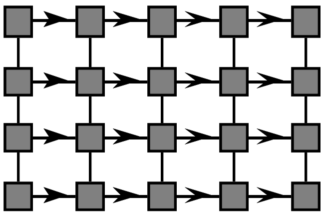
```
LOOP OF grid DO
  ...
  IF x>0 THEN EXIT END;
  ...
END; (* loop *)
```

In case one would like to perform an operation on *all PEs* inside a nested selection or loop, the ALL block may be used to re-activate all PEs of a configuration.

```
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;
```

## 1.6. Structured Data Exchange

Data exchanges between processors can be accomplished with simple symbolic names, thanks to the network declaration described earlier. Data exchange of a local vector variable between all or just a group of PEs can be invoked by calling system function MOVE with the name of a previously defined connection. Only active PEs participate in a data exchange operation. Figure 1.8 shows an example of a data exchange in the grid structure defined earlier. The expression returns vector variable x shifted one step (one PE) to the east.



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

**Figure 1.8** Synchronous data exchange

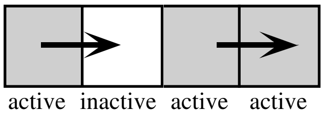
For the data exchange operation shown above, sender-PE *and* receiver-PE of a data exchange have to be active. For the operations SEND and RECEIVE shown below, it is sufficient for *only* the sender (or *only* the receiver, respectively) to be active. These operations are especially needed for the data exchange between different topologies, since due to the SIMD model only one PE structure can be active at a time. Unlike the other data exchange operations, SEND is a procedure (not returning a value) and therefore takes two arguments, first the expression to be sent, and second the variable to receive the expression.

```
SEND.east(4*x, y);
y := RECEIVE.north(x);
```

The following comparison shows the differences in data exchange operations by using a simple list topology. In figures, an arrow represents data transport, while a white space marks an inactive PE. Let us assume that for the context of the following data exchange operations that all PEs are active but one (e.g. the data exchange might occur inside an `IF`-selection, which deactivated one of the PEs).

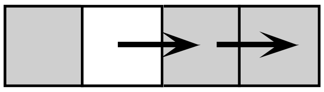
```
CONFIGURATION list[1..max];
CONNECTION right: list[i] -> list[i+1];
VAR x,y: list OF INTEGER;
```

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



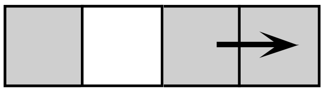

| Example: | before | after |
|---|---|---|
| | x: `1 2 3 4` | x: `1 2 3 4` |
| | y: `0 0 0 0` | y: `0 1 0 3` |

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



| Example: | before | after |
|---|---|---|
| | x: `1 2 3 4` | x: `1 2 3 4` |
| | y: `0 0 0 0` | 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 | after |
|---|---|---|
| | x: `1 2 3 4` | x: `1 2 3 4` |
| | y: `0 0 0 0` | 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` has to be a procedure for it writes to inactive PEs, while `MOVE` and `RECEIVE` are functions returning vector data.
- All data exchange operations shift data in the direction indicated by the connection identifier (here: `right`). This also holds for the `RECEIVE` function, so it does **not**

have the semantics "receive *from* a direction" (e.g. `RECEIVE.right` shifts data to the right, not to the left).

Additional data exchange modifiers may be specified for some of the data exchange operations. One is for multiple data movement steps and one is for an implicit reduction; both are separated by colons.

A data exchange operation may be executed several steps (*subsequently*) in a specified direction for the `SEND` statement. After each step, all PEs that received data will be active, sending data in the next step. In case one data element is sent to multiple PEs, then all of these will be active in the next step. This feature is extremely useful when exchanging data over a tree structure.

```
VAR x,y: list OF INTEGER;
...
SEND.right:2 (x,y);               move data two steps to the east
```

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

| | *before* | *copy expr.* | *1st step* | *2nd step* | *assignment* |
|---|---|---|---|---|---|
| x: | **1 2** 3 4 | – | – | – | 1 2 3 4 |
| intermediate variable: | – | **1 2** 3 4 | 1 **1 2** 4 | 1 1 **1 2** | **–** |
| y: | 0 0 0 0 | – | – | – | 0 0 1 2 |

A problem might occur, if several PEs are connected to a single one (many-to-one connection). There are two possibilities to avoid an undetermined result (any of the incoming data values could be chosen). One can either deactivate unwanted PEs (`IF`-selection), so they cannot participate in a data exchange, or one can use a reduction function with the data exchange.

For example, in the tree network shown before, one might want to send only the left children's data to the parents and discard the right children's data:

```
VAR u,v,w: tree OF INTEGER;
...
v := u;                           initializes all components of v
IF EVEN(ID(tree)) THEN            moves only the left children's data up the tree
  SEND.parent (u,v)
END;
```

Now assume, one does **not** want to discard information, but one would like to add the left and right child's data before sending to the parent node:

```
w := MOVE.parent:#SUM (u);        moves data from both children to the parent,
                                  resolving multiple arriving data by adding
```

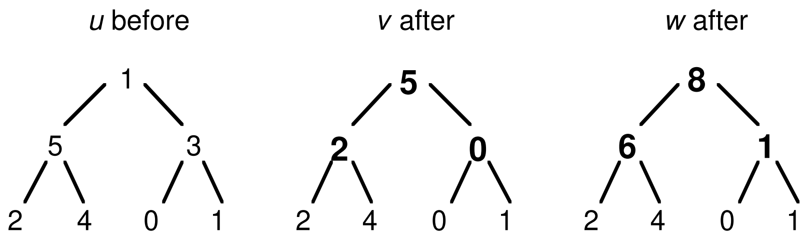



**Figure 1.9** Move without and with reduction

There is a number of system-defined operations to do this *reduction*, and also user-defined operations may be specified (see section on reduction below).

Configuration boundaries often cause trouble in SIMD programming, for they frequently require special treatment to avoid undefined data. This is not the case for Parallaxis. Here, it is allowed to send data off the edge and try to receive data from beyond the edge of a configuration. After initializing the send expression with the vector parameter value supplied, data sent offside a configuration is deleted, while an attempt to read from beyond leaves the particular PE's data unchanged. Therefore, within the same configuration, undefined values cannot occur in a data exchange operation.

## 1.7. Unstructured Data Exchange

Structured data exchange makes application programs easy to write and understand. In some cases it also makes them faster, when better use can be made of the physical connection structure of a particular parallel system. However, in some cases it may be desirable to perform an unstructured data exchange. This reflects an arbitrary permutation of the components of a vector variable, which may be difficult to write down using only structured data exchanges with user-defined connections.

For example, each component of a two-dimensional vector (a matrix) is to be sent to a destination address, which is being computed at run-time. When only structured data exchange, e.g. via a grid, is possible, one has to program a communication procedure, which shifts the matrix elements in several steps over the grid. This approach will work, however, some parallel computer systems (like the MasPar MP-1 and MP-2 [MasPar 91] and the Connection Machine CM-2 [Thinking Machines 89]) have a global connection structure, which allows an arbitrary unstructured data exchange. In this case, specifying direct destination addresses for each component of a vector variable may result in a faster program. Please note, that despite the availability of specialized commands for unstructured data exchange, execution of those may be quite expensive. For example, a grid operation at the MasPar MP-1 requires about the same time as a simple arithmetic operation (addition), but a non-grid data exchange takes about 100 times longer to execute. Please note that the unstructured data exchange is nevertheless a machine-independent operation. If a certain SIMD architecture does not provide a general communication structure, then this data exchange will be routed transparently over the simpler network provided (e.g. a grid or a ring) taking several execution steps.

In Parallaxis, the `SEND` and the `RECEIVE` operations may take an index expression instead of a connection name. As before, when using `SEND`, only active PEs send data, and when using `RECEIVE`, only active PEs receive data. However, these two operations differ in their index semantics, as is shown for an example in Figure 1.10. In order to avoid confusion, operation `MOVE` may not be used with an index expression.

```
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

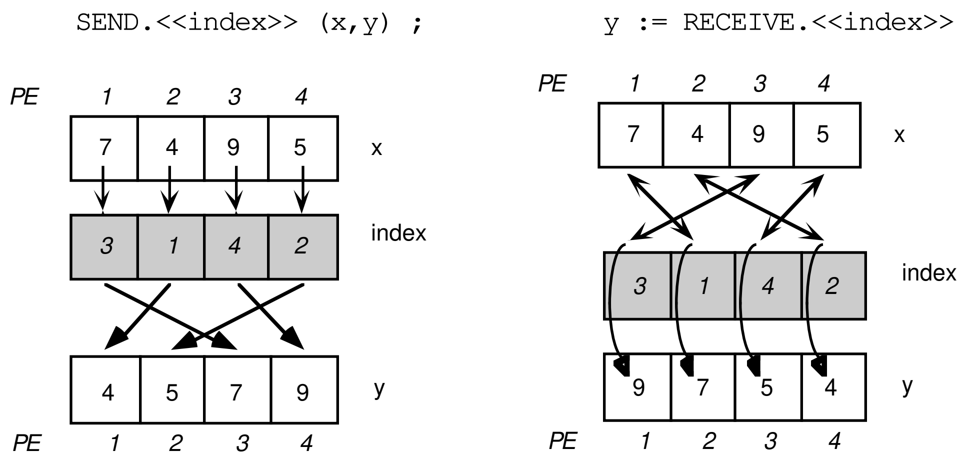



**Figure 1.10** Unstructured data exchange

Besides using a single index, referring to the `ID` position of PEs, several indices referring to `DIM` positions may be used as well. Also, this kind of data exchange does not have to be a one-to-one correspondance. If several indices refer to the same PE position, for `RECEIVE` (one-to-many) this results in a broadcast, while for `SEND` (many-to-one) an arbitrary component is selected – unless a reduction operation (see section on reduction) is specified for resolving collisions.

```
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);
```

in case the expression `index` does not provide a 1:1 permutation, it may be desirable to perform a reduction of multiple values arriving in one port, in order to avoid the assignment of an arbitrary one of these; positions not indexed get the sender's original elements

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

Two kinds of abbreviations are possible for data exchanges with index expressions:

a) If the positions in one dimension are to remain unchanged, one should use the expression `DIM(`*conf_name*`,` *dim_no*`)` as an index. This may be abbreviated with the symbol '*'.

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

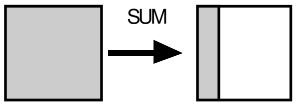
is equivalent to:

```
SEND.<:*,d1:> (x,y) ;
```

b) If a dimension is to be collapsed, a many-to-one data exchange may be used in combination with a reduction. For example, a matrix may be collapsed to a single column by reducing all of its rows. This can be done by sending all row elements to the first element in the same row with a reduction operation. This may be abbreviated by using a reduction operation (e.g. `#SUM`) instead of an index expression. Only a *single* reduction may be specified in an index expression.

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

`SEND.<:d2,#SUM:> (x,y);` dimension 1 is reduced by addition (rows accumulate data in their first positions) with permutation in dimension 2 according to `d2`

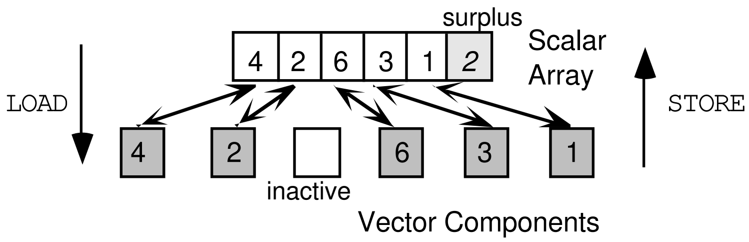



E.g. $x = \begin{pmatrix} 7\ 3\ 5 \\ 4\ 2\ 3 \end{pmatrix}$ $d2 = \begin{pmatrix} 1\ 1\ 1 \\ 2\ 2\ 2 \end{pmatrix}$ **then** $y = \begin{pmatrix} \mathbf{15}\ 3\ 5 \\ \mathbf{9}\ 2\ 3 \end{pmatrix}$

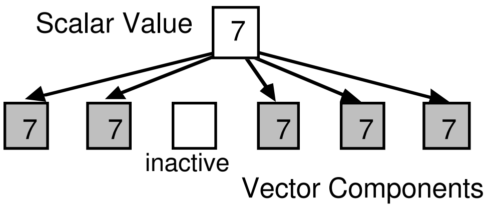
## 1.8. Exchange between Scalar and Vector Data

Communication between the control processor and the parallel PEs also requires additional language constructs or in some cases an adapted semantics. Transferring a scalar field into a parallel vector is invoked with procedure `LOAD`, while transferring data back into a scalar field from a vector is accomplished with `STORE` (see Figure 1.11). Only active PEs participate in this sequential data exchange. `STORE` with inactive PEs does not result in gaps in the scalar array, but data elements are stored subsequently. `LOAD` with inactive PEs assigns the next array value to the next active PE, no scalar array elements will be skipped. Surplus elements will not be used, too few elements leave the corresponding array elements (or vector components, respectively) unchanged. The execution of this operation usually requires *n* time steps for a data array with *n* elements. A scalar integer variable may be specified as an optional third parameter for `LOAD` and `STORE`, which limits the number of data items transferred and also receives the number of data items actually transferred after the operation.

```
CONFIGURATION list[1..n];
VAR  s: ARRAY[1..n] OF INTEGER;
     t: INTEGER;
     v: list OF INTEGER;
...
LOAD (v, s);  (* from scalar to vector *)
STORE(v, s);  (* from vector to scalar *)
STORE(v, s, t);  (* here, t becomes n  *)
(* require n steps each *)
```



```
v := t;  (* requires 1 step *)
```




takes 1 step

**Figure 1.11** Data exchanges between PEs and control processor

Figure 1.11 (bottom) also shows an assignment in which a (constant or variable) scalar data value is copied into all or a group of PEs. Every component of the vector contains the same value as the scalar. This operation is implemented by an implicit *broadcast* and therefore requires only a single time step.

## 1.9. Reduction

The reduction of a vector to a scalar is another important operation. The `REDUCE` operation handles this task in conjunction with a system-defined or user-defined (programmable) reduction operation (see Figure 1.12). System-defined operators are:

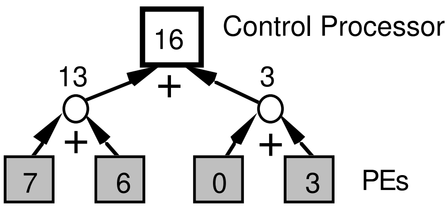
```
SUM, PRODUCT, MAX, MIN, AND, OR, FIRST, LAST
```

The operators `FIRST` and `LAST` return the value of the first or last currently active PE, respectively, according to its identification number (`ID`).

Example: `s    := REDUCE.FIRST(x)`    is identical to:

```
pos := REDUCE.MIN(ID(x));
s   := x<<pos>>;
```

All other reduction operators' functions can easily be deduced from their names. The execution of a reduction operation requires about $\log_2 n$ time steps for a vector with $n$ active components. However, this time estimation depends on the physical connection structure of the PEs.




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

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

**Figure 1.12** Vector reduction in Parallaxis

The following example shows the use of the `REDUCE` operation with a user-defined function. Such a function has to have two vector input parameters and has to return a vector value of the same type. Note that the reduction function implemented by the user should be associative and commutative, or unpredictable results may occur,
e.g. $(1 - 2) - 3 \neq 1 - (2 - 3)$.

```
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);
```

There are a few places, where substituting a scalar constant in lieu of a vector variable makes sense, but lacks information about the configuration to be used. Consider the problem of counting the number of active PEs for some configuration. Instead of using a vector variable, the constant 1 can be used for each PE, however, it has to be type cast to the appropriate configuration:

```
s := REDUCE.SUM( grid(1) );
```

## 1.10. Modules

In Parallaxis, like in Modula-2, each module consists of two files: a definition module and an implementation module. The only exception is the main module, which starts a program and does not have a definition module. Parallaxis also offers a `FOREIGN` module, which serves as the definition part for linking routines written in another language.

```
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.
```

The sample module constellation above shows declaration, export, and import of the user-defined procedure `myproc`. It is exported in the definition module together with its procedure head, showing number and type of parameters. The actual implementation of `myproc` is hidden in the corresponding implementation module (this is also used for including routines from different programming languages). If an export list is missing in a definition module, then its whole contents is being exported.

The module importing and using procedure `myproc` in the example is the main module (it has neither keyword `DEFINITION` nor `IMPLEMENTATION`).

There are two ways of importing objects. The first imports *individual objects* from a module, the second includes *all objects* from a module (in that case, however, an object name always requires the module name as a prefix, in order to avoid name conflicts):

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

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

## 1.11. Input and Output

I/O operations in Parallaxis are very similar to Modula-2. However, in Parallaxis they are "built-in" and need not be explicitly imported as in Modula-2. This change was required, since read- and write-operations in Parallaxis may take either scalar or vector arguments.

The major text input and output operations in Parallaxis are shown below (ASCII data). I/O operations do *not* generate run-time errors (e.g. in case of inappropriate data to be read or in case of insufficient disk space during a write operation). Instead, the scalar boolean variable `Done` is set according to the success of the I/O operation. This variable can be checked to perform appropriate actions, in case its value is `FALSE`. When reading strings, integers, cardinals, reals, or booleans, the scalar character variable `termCH` is set to the next unread character in the input stream, which was responsible for terminating the read operation. `EOL` is a constant of type character, which stands for the system-dependent end-of-line character.

| | |
|---|---|
| `WriteLn;` | Start a new line (no arguments) |
| `Write(c);` | Write character *c* |
| `WriteString(s)` | Write string *s* |
| `WriteInt(i,l);` | Write integer *i* using *l* print spaces<br>(right adjusted: add blanks in front of number) |
| `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,l);` | Write boolean *b* using *l* print spaces<br>(left adjusted: add blanks after string) |
| | |
| `Read(c);` | Read character *c* |
| `ReadString(s)` | Read string *s* |
| `ReadInt(i);` | Read integer *i* |
| `ReadCard(c);` | Read cardinal *c* |
| `ReadReal(r);` | Read real *r* |
| `ReadBool(b);` | Read boolean *b* |
| | |
| `OpenOutput(s);` | Open file with name *s* for writing<br>(following write operations write to file) |
| `CloseOutput;` | Close file, redirect output to *stdout* |
| `OpenInput(s);` | Open file with name *s* for reading<br>(following read operations read from file) |
| `CloseInput;` | Close file, redirect input back to *stdin* |

The following gives a number of sample write operations; read operations work the same way.

```
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 *)
```

## 1.12. Control Structures

Sequential control structures are also identical to Modula-2. Please note, that in Modula-2 (unlike Pascal) all control structures have to be terminated with keyword `END` (with the exception of `REPEAT`, which has a different terminating keyword) and all procedures/functions repeat the procedure name with the final `END` (see the following examples). All control structures 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);
(* reference parameter *)
BEGIN (* procedure *)
  ...
END abc;
```

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

## 1.13. Relation to Modula-2

Parallaxis is a true extension of sequential Modula-2, with one exception. No nesting of local modules is allowed as an implementation restriction.

Besides this limitation, there are also some sequential extensions to the Modula-2 syntax. First, a comparison may have an arbitrary number of expressions. This allows range checks not possible in Modula-2, like the following:

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

Next, the power operator '`**`' has been included. It is also possible to specify constant records or constant arrays by using the type name as a function identifier (only for record and arrays of simple types). This is a useful feature for initializing record or array variables:

```
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);
```

Like in the programming language C, the number and string contents of the command line parameters can be read in Parallaxis:

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

Mathematical and I/O operations are pre-defined in Parallaxis and do not have to be explicitly imported, as in Modula-2. This is necessary, since all the procedures and functions may take either scalar *or* vector parameters.

There is some confusion about the `MOD` operator in Modula-2 and C implementations. In Parallaxis, the result of a modulo operation will always be positive, especially an expression like `(-1) MOD 5` equals `4` in Parallaxis (as does `1 MOD (-5)`), and not `-1` as in C. This feature is heavily used in connection spefications, e.g. to construct a torus:

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

## 1.14. Efficiency

Naturally, some Parallaxis statements require more computation time than others, while most of these comments apply only to the generation of parallel code for a SIMD system and not for the simulation on a single-processor workstation. The following summarizes remarks made throughout the text.

1. PE Data Exchange
   structured data exchange may be faster than unstructured data exchange
   - 1.a especially a structured data exchange on a grid may be much faster
     (if it corresponds to the hardware, e.g. 100 times on a MasPar MP-2 system)
   - 1.b a structured data exchange on a torus may be as fast as on a grid, only if the torus size equals a possible torus in hardware
   - 1.c using the open configuration construct will cause some run-time overhead, since the connections have to be initialized
2. Reduction
   - 2.a reduction operations require $\log_2 n$ steps for $n$ PEs, each step comprising a data exchange and an arithmetic operation (in general this data exchange will be unstructured)
   - 2.b using a data exchange with reduction may be very expesive and should therefore be avoided if possible (connection groups may have to be executed sequentially)
3. Front End Data Exchange
   data exchange to and from the front end (load and store), as well as read and write operations with vector data require $n$ steps for $n$ PEs (sequential execution).

4. Broadcasting
   - 4.a broadcasting a single value from the front end to all PEs requires only 1 step.
   - 4.b broadcasting a single value from one PE to all other PEs requires 2 steps (1 from PE to front end plus 1 from front end to all PEs)

# 2. Sample Algorithms

The following sections demonstrate a few sample algorithms programmed in Parallaxis-III.

## 2.1. Image Filtering

The Laplace operator is one possible operator for emphasizing edges in a gray-scale image (edge detection). The operator carries out a simple local difference pattern (see Figure 2.1) and is therefore well suited to parallel execution. The Laplace operator is applied in parallel to each pixel with its four neighbors.

For each pixel:

| | | |
|---|---|---|
| | –1 | |
| –1 | 4 | –1 |
| | –1 | |

In the whole image:

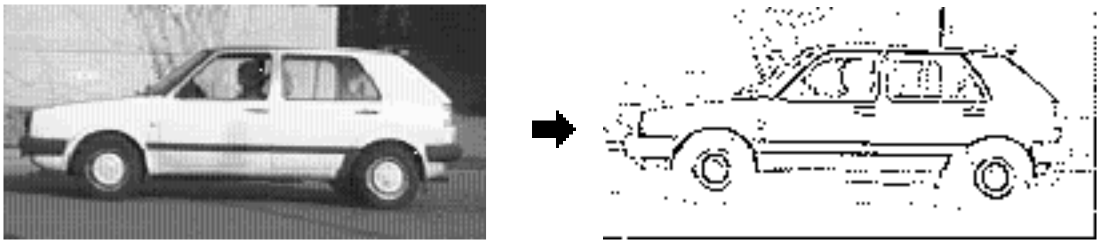


**Figure 2.1** Edge detection with Laplace operator and threshold

A procedure for the Laplace operator in Parallaxis, using the grid structure defined earlier, looks like the following:

```
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;
```

The data from neighbor PEs is obtained by local data exchange operations, which are directly used in the arithmetic expression to be returned. This simple procedure does not handle range limits for gray-scale values, e.g. `[0..255]`.

## 2.2. Cellular Automata

Cellular automata are likewise an application area well suited to SIMD systems. Every cell can be assigned a processor and carries out the same processing instructions. One of the most prominent cellular automata is Conway's ‘Game of Life’, which is a two-dimensional structure changing in time. The cellular automaton shown here is simpler and only one-dimensional; however, it generates a two-dimensional image during execution (one line on every iteration). The processing instructions are conceptually simple: Every cell has only two possible states and computes its successor state from an exclusive-or of the states of its left and right neighbor. The middle cell is initialized with `TRUE` (printed as ‘`X`’), while all of the other cells are initialized with `FALSE` (printed as empty spaces).

```
MODULE cellular_automaton;
CONST n = 79;            (* number of elements        *)
      m = (n+1) DIV 2;   (* number of loops := middle *)
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;
            (* = ARRAY[FALSE..TRUE] 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.
```

The `CONFIGURATION` and `CONNECTION` declarations define a doubly linked list of PEs. For screen output of the current state of all PEs, every boolean state is converted to a symbol of type `CHAR`. The complete character vector is then written to the screen sequentially by a vector valued `Write` operation. The initialization of the main program assigns the value `TRUE` only to the middle PE with number (`n+1 DIV 2`); all other PEs receive the value `FALSE`. Finally, there is a scalar loop which prints the current state and calculates the next cell state in each pass, using data exchange with the left and right neighbors.

Figure 2.2 shows the states of this cellular automaton, in which time progresses from top to bottom.

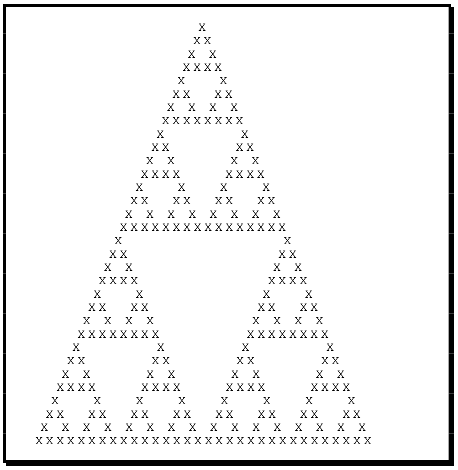


**Figure 2.2** Output of cellular automaton

## 2.3. Generation of Fractals

The algorithm presented here is a problem that can be solved by a data parallel program with the 'divide-and-conquer' method. This is, however, not possible with all divide-and-conquer algorithms, since the different branches usually have to carry out different program parts. The algorithm presented here generates a one-dimensional fractal curve using midpoint displacement (see [Peitgen, Saupe 88]). It starts with a straight line, which has its midpoint displaced up or down according to a weighted random value. Two line segments with different slopes arise from this process, and these are handled recursively in parallel in exactly the same way during the following steps (see Figure 2.3). The number of line segments to be processed doubles at every step, until the required resolution is reached. The processor structure used here is a binary tree structure. Beginning with the root, in each step the following tree level is activated until the leaves are reached. After computing the leaf level, the program prints the result values of the whole tree and terminates. Since with the exception of the communication there is *only one* tree level active at a time, only half as many PEs would be sufficient with a more complicated connection structure.

A simple tree structure is declared in the Parallaxis program, through which start and end points of the lines are passed ('`**`' denotes the exponential operator). Only procedure `MidPoint` is shown, where the actual processing takes place. The tree levels are activated successively, and the midpoint displacement is carried out for each level (using function `Gauss` not shown here, which returns a gaussian distributed random value). While the leaves have not yet been reached, the data values `low` and `high` are passed to the child nodes. In this method, the left child receives the values `low` and `x` as the start and end points of its line segment, while the right child receives `x` and `high`.

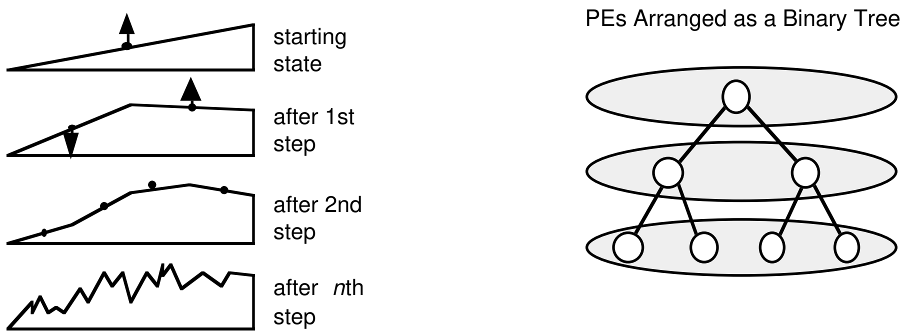



**Figure 2.3** Divide-and-conquer implemented with tree topology

```
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;
```

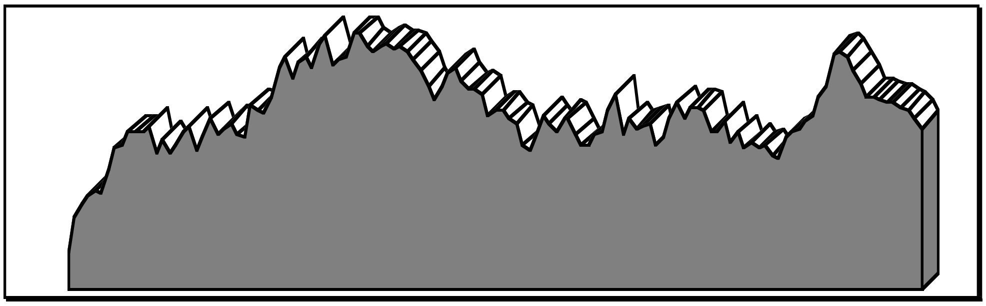


**Figure 2.4** Fractal curve generated by program

The computation time required for this program is $\log_2 n$ steps for $n$ leaf nodes (equal to the tree's height). Figure 2.4 shows a fractal curve generated by this program with 127 PEs, while Figure 2.5 shows the beginning of another fractal number sequence interpreted as fractal music.

**Figure 2.5** Fractal music

## 2.4. Sorting

A number of different SIMD algorithms exist for the problem of sorting. The ‘odd-even transposition sort’ (OETS) is presented here as a representative, which can be understood as a parallel version of bubble-sort. OETS sorts $n$ data elements with $n$ PEs in $n$ steps. Figure 2.6 shows the progression of the parallel algorithm. Every PE holds one of the numbers to be sorted. During processing, the algorithm differentiates between odd and even steps. In odd steps, all of the PEs with odd identification numbers compare their element values with that of their right neighbor (PEs: 1–2, 3–4, 5–6, etc.) and carry out a data exchange if the value of their own element is larger than that the one of their neighbor. In the even steps, all of the PEs with even identification numbers carry out an analogous comparison and, if necessary, a data exchange with their right neighbor (PEs: 2–3, 4–5, 6–7, etc.). After $n$ iterations, the list is sorted. The right part of Figure 2.6 visualizes the algorithm, with each line in the image representing the current state of the list of PEs, while the time is progressing from top to bottom. Each pixel in a line is represented by a gray value according to its data value (low values are dark, high values are bright). In the beginning (top line), the gray values (data values) are unsorted, while they become slowly disentangled during progression of the algorithm, until the bottom line contains the gray values in perfect order. The movement of individual cells to the left or right reminds of bubble-sort.

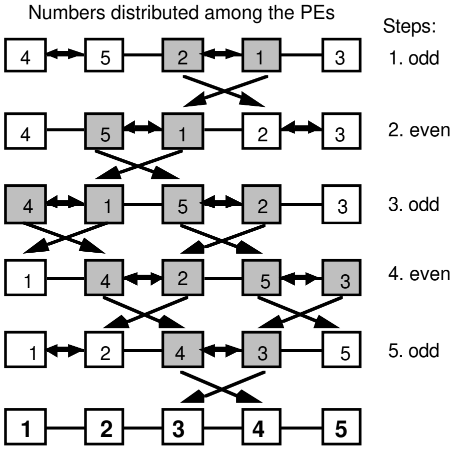



**Figure 2.6** Example of odd-even transposition sorting

In the Parallaxis program, first the variable `lhs` determines whether a PE is in the role of the left or the right partner of a comparison. This role changes for each loop iteration. During each pass, the PEs get the data values of their left and right neighbor in `l` and `r`, respectively. The partner on the left-hand side uses the right neighbor's value for comparison, while the partner on the right-hand side uses the value of its left neighbor. The complicated comparison '`lhs = (comp<val)`' is true for the left PE partner when the right comparison value is greater than its own value; it is **also** true for the right PE partner, if the left comparison value is smaller than its own value. So a single key comparison is sufficient for all PEs to find out where swapping of data values are required. By applying a more complex compound topology, the program could be optimized further, such that only a single data exchange operation would be required for each pass through the loop.

```
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));  (* PE is left-hand-side of comparison *)
  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 & (comp< val) *)
    lhs := NOT lhs;                        (* or rhs & (comp>=val) *)
  END;
  WriteInt(val,5);
END sort.
```

Figure 2.7 shows the runtime behaviour of the OETS sorting algorithm for sorting of 1000 numbers. The PE load curve shows a continuous high processor utilization.

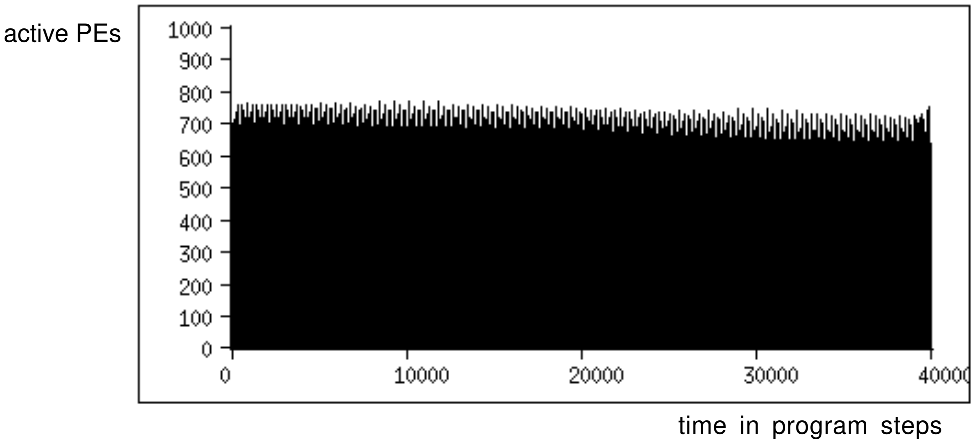



**Figure 2.7** Processor utilization of the sorting algorithm

## 2.5. Stereo Image Analysis

Spatial vision is a fascinating phenomenon, especially considering that this activity which people take for granted can only be implemented with huge computational costs on computer systems. However, the fundamental stereo image analysis algorithms can be easily translated into synchronous parallel programs. The three-dimensional impression in human vision comes from the fact that the left and right eyes see images that have slight differences due to the slightly different viewing angle of each eye (see Figure 2.8: Line *A*'*B*' appears longer to the left eye than line *A*"*B*" appears to the right eye.). The displacement of points in the left and right image is known as ‘disparity’ and can be used to determine the difference in distance from the observer of the corresponding points in the physical world. Unfortunately, it is not very easy to match points in the image of the left eye with points in the image of the right eye.

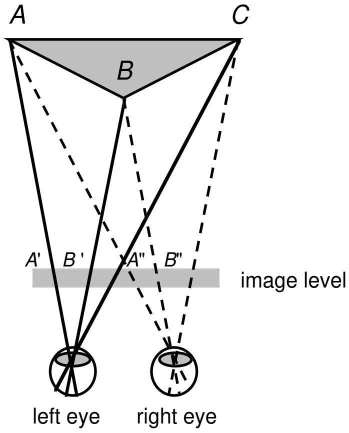
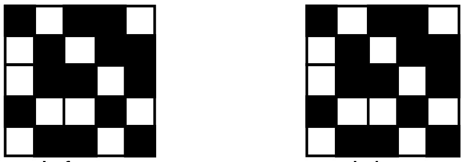



**Figure 2.8** Stereo vision

Quite some years ago, Julesz used ‘random dot stereograms’, to show that the brain is able to interpret stereograms, with each monocular image by itself lacking any image contents (see [Julesz 60] and [Julesz 78]). Each of the left and right images considered alone appears to be completely randomly generated and contains no information. The 3D effect emerges from the interaction of the two images inside the brain, where the stereo disparity in the random patterns are analyzed and converted into spatial sensations. Random dot stereograms are very easy to generate on a computer system and do not have the disadvantage of photographs, such as imprecision and alignment problems. Neither do they require edge detection, since random dot stereograms consist of edges only.

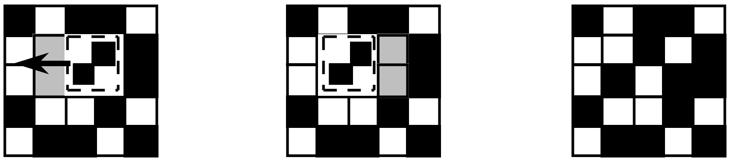
**Generation of random dot stereograms**

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



left right

2. *Raising or lowering areas*



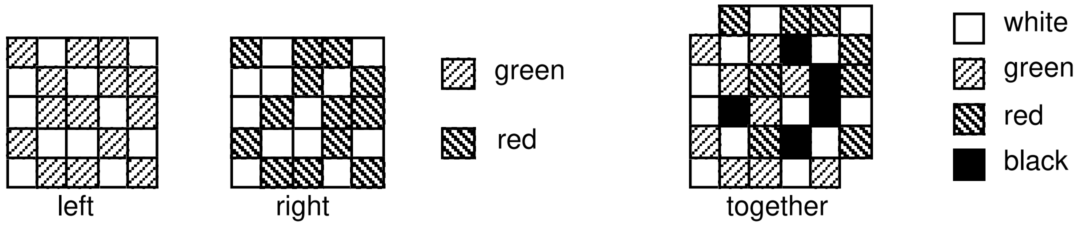
right image before right image after right image filled

An object can be raised to a certain altitude by shifting it only in the **right** image a corresponding number of pixels to the **left** (or lowered to some depth level by shifting to the right). The gap created in this process is then filled with a random dot pattern. *The left image remains unchanged*. This step is repeated for every area that will be elevated or lowered from the original image plane.

```
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**

There is a quite a number of possible techniques for viewing stereograms, all of which ensure that only the left picture is seen by the left eye and only the right picture by the right eye. Examples are prisms, LCD shutter displays, polarization filters, red/green filters, head mounted displays (two screens) or simply the concentrated viewing of each image with one eye. Red/green glasses (*anaglyph method*) represent the easiest option to implement. First the left image is colored green/white and the right image is colored red/white. Then the images are printed on top of each other, but slightly out of alignment, so that the displacement cannot be recognized without the glasses. If two colored pixels are to be printed on top of each other, they should appear on paper or on the screen as a black pixel.




The red filter in front of the left eye only allows the red light to pass through and therefore makes the green and black points appear dark and the white and red points light. This corresponds exactly to the original left image. The green filter in front of the right eye only allows the green light to pass through, therefore the red and black pixels appear dark. The green image is thereby filtered out of the mixed image stereogram (see Figure 2.9).

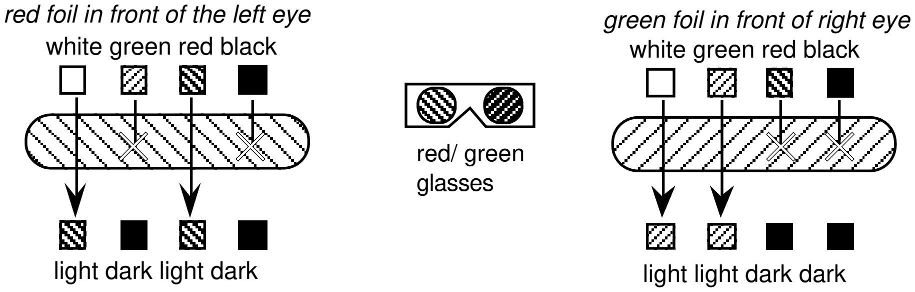
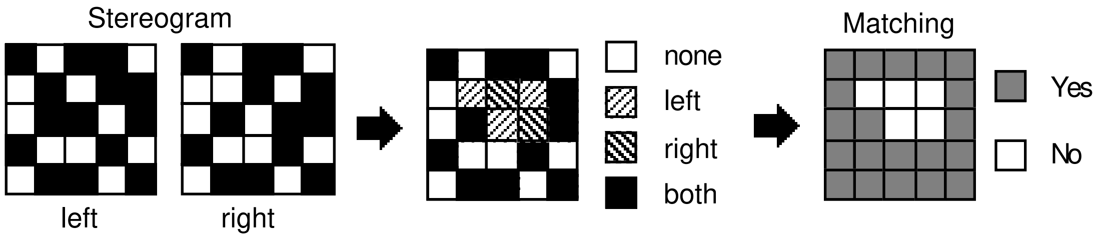



**Figure 2.9** Effect of stereo glasses

The ensuing reverse calculation from the stereograms (e.g. regenerating the depth information from the left and right images) is much more difficult and computationally expensive than the generation of the stereograms. In addition, the reverse calculation can never be 100% correct. Usually, there are individual points which are assigned false depth values.

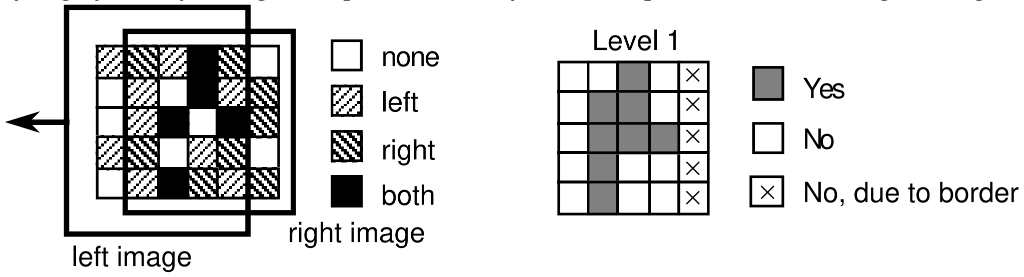
**Analysis of random dot stereograms**
(Reverse calculation of depth information from stereograms)

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



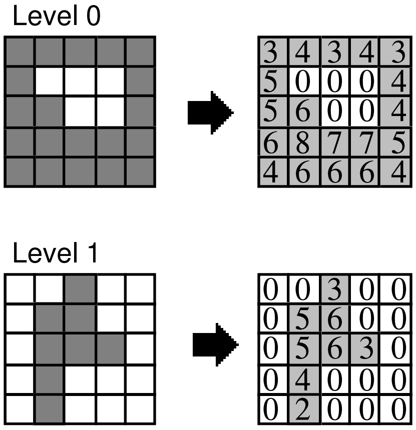

A pair of corresponding pixels (one pixel from left and right image each, at the same position) match if both pixels are black or both are white. This operation can be performed in parallel on all pixels.

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



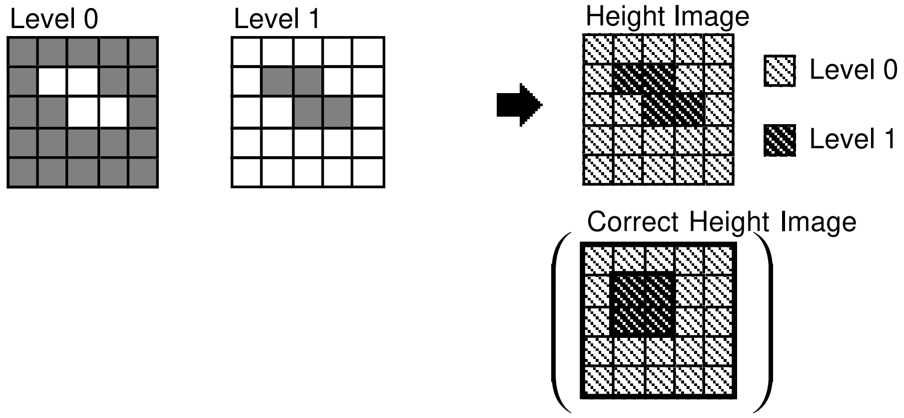

This shift step with subsequent comparison of left and right image (see step 1) is carried out iteratively for each depth level. This gives the matching pixels for all height levels.

3. *Determining the depth of each Pixel*




This step is also carried out iteratively for each depth level, but in parallel for all pixels. Data from a $5 \times 5$ local neighbor field (only a $3 \times 3$ field is shown in the figure) with eight-way grid connections is used to determine the matching neighborhood for all pixels. Pixels not matching receive the value zero, while pixels matching receive the sum of all neighbor pixels that also match, plus one for themselves.

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




The level (depth) that gives the largest matching sum for each pixel is selected as the level for that pixel (in the case of equal values, the lower level is selected). If the neighborhood values for all levels are zero, the level of one of the neighbor pixels is chosen. This computation can be done in parallel for all pixels.

The data from all the levels can be combined into a depth image ('false color image'), as shown here. For each pixel, its color represents its depth. The solution found in the example is not totally correct, since differences of isolated pixels are not avoidable with this procedure. The correct depth image is shown in parentheses below it.

*5. Filter (optional)*

In order to eliminate single pixel errors, a local $3 \times 3$ matrix can be used in each level, to sum the number of pixels already in that level. Pixels having too few neighbors in this level (number < threshold) are assigned to another level. The filter operation can be performed in parallel for all pixels.

The algorithm for analyzing random dot stereograms shown here can very easily be translated into a data parallel program. Function `sum_3x3` returns the sum of a local 3×3 neighborhood.

```
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(v) + MOVE.right(v); (* horiz. sum   *)
    RETURN( l_r + MOVE.up(l_r) + MOVE.down(l_r) ); (* vert. sum *)
  END sum_3x3;                                  (* cumulative *)

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;
```

Figure 2.10 shows a sample stereogram and the corresponding calculated depth information (containing a few inevitable defects) together with a subsequent smoothing using a 3×3 median filter operation.

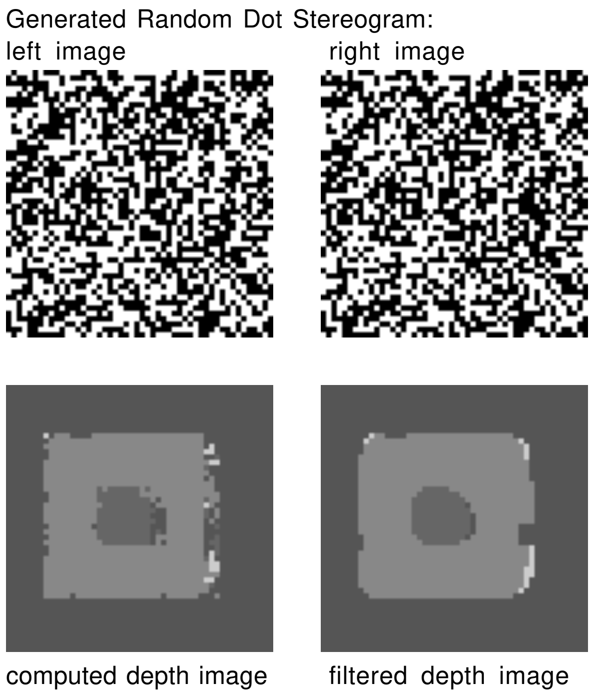



**Figure 2.10** Random dot stereogram with calculated depth information

Figure 2.11 shows the generation of a random dot stereogram with 100×100 pixels and the subsequent computation of the corresponding depth image. The first half of the diagram corresponds to the stereogram generation. The PE-activation is decreasing, since for this sample run the image area being shifted is decreasing as well. The second half of the diagram shows a continuous high processor utilization, while individual peaks correspond to the iterations of the main analysis loop, comprising steps 2 to 4 of the algorithm above.

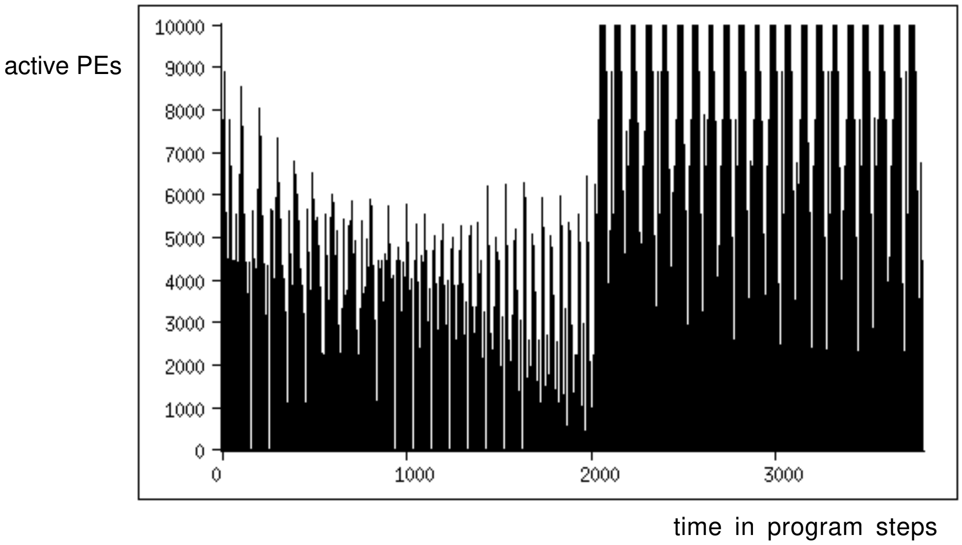



**Figure 2.11** Processor utilization of random dot stereogram program

# 3. Acknowledgements

I would like to thank Eduard Kappel, Hartmut Keller, and Harald Lampke, who implemented Parallaxis-III at the Univ. Stuttgart, gave valuable suggestions on the Parallaxis-III syntax and semantics, and proof-read several drafts of this paper. Thanks to Panagiotis Metaxas, who made numerous comments for improvement. The work of Ingo Barth, Frank Sembach, Karsten Krauskopf, and Stefan Engelhardt on Parallaxis versions 1 and 2 is also acknowledged. Compiler and simulation environment for Parallaxis are available as public domain software from the Univ. Stuttgart via 'anonymous ftp'.
Server: `ftp.informatik.uni-stuttgart.de` (129.69.211.2)
Directory: `pub/parallaxis`

# 4. Literature

[Barth, Bräunl, Engelhardt, Sembach 92] I. Barth, T. Bräunl, S. Engelhardt, F. Sembach, *Parallaxis Version 2 User Manual*, Second Edition, Computer Science Report, no. 2/92, Univ. Stuttgart, Feb. 1992, pp. (110)

[Bräunl 89] T. Bräunl, *Structured SIMD Programming in Parallaxis*, Structured Programming, vol. 10, no. 3, July 1989, pp. 121-132 (12)

[Bräunl 91] T. Bräunl, *Designing Massively Parallel Algorithms with Parallaxis*, Proceedings of the 15th Annual International Computer Software & Applications Conference, compsac91, Sep. 1991, pp. 612-617 (6)

[Bräunl 93] T. Bräunl, *Parallel Programming – An Introduction*, Prentice Hall, Englewood Cliffs NJ, 1993

[Julesz 60] B. Julesz, *Binocular Depth Perception of Computer Generated Patterns*, Bell Systems Technical Journal, no. 38, 1960, pp. 1001-1020 (20)

[Julesz 78] B. Julesz, *Cooperative Phenomena in Binocular Depth Perception*, Sensory Physiology, no. 8, 1978, pp. 215-252 (38)

[MasPar 91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) User Guide*, Software Version 2.2, MasPar System Documentation, DPN 9302-0101, Dec. 1991

[Peitgen, Saupe 88] H.-O. Peitgen, D. Saupe (Eds.), *The Science of Fractal Images*, Springer-Verlag, Berlin Heidelberg New York, 1988

[Thinking Machines 89] Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, version 5.1, Technical Report, May 1989

[Wirth 83] N. Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelberg New York, 1983

# Appendix

## A.1. Data Types

Parallaxis supports all data types available in Modula-2:

| | |
|---|---|
| `INTEGER` | integer numbers (4 bytes) |
| `CARDINAL` | integer numbers greater or equal to 0 (4 bytes) |
| `REAL` | floating point numbers (4 bytes) |
| `CHAR` | character values (1 byte),<br>ordinal values ranging from 0 to 255 |
| `BOOLEAN` | truth values (1 byte), equal to `(FALSE,TRUE)` |
| `ARRAY .. OF ..` | array of data values |
| `RECORD .. END` | collecting several entries to a structured type |
| `SET OF ..` | set of a simple type |
| `BITSET` | equivalent to `SET OF [0..31]` |
| `POINTER TO ..` | pointer to a type, data has to be allocated dynamically |
| `PROCEDURE (..)` | procedure or function type,<br>e.g. `TYPE function = PROCEDURE (REAL): REAL;` |
| *enumeration* | enumerating all values of a type (1 byte)<br>e.g. `TYPE day = (mo, tu, we, th, fr, sa, su);` |
| *subrange* | limited range of values (from 1 to 4 bytes)<br>e.g. `TYPE workday = [mo..fr];`<br>`digit   = [0..9];` |

## A.2. Built-in Functions and Procedures

All built-in functions and procedures may be called with either scalar or vector arguments. The return value of such a function is scalar if its argument is scalar, and vector if its argument is vector. Type `INTEGER` always includes type `CARDINAL`.

In Modula-2, mathematical functions and I/O-procedures are separate modules and their objects have to be imported. In Parallaxis, however, all these functions and procedures can be used with either scalar or vector arguments. Therefore, they had to be included as built-in functions and procedures.

### General Functions

| | |
|---|---|
| `FLOAT(i)` | return real value of integer *i* |
| `TRUNC(t)` | return value of real *r*, truncated to an integer |
| `ABS(i);` | return absolute value of x, type Integer or Real |
| `CHR(i)` | return character with ordinal number *i* |
| `VAL(t,i)` | return value of type *t* with ordinal number *i* |
| `ORD(c)` | return ordinal number of *c* for character or enumeration type, starting with 0 |
| `MAX(t)` | return maximum value of type *t* |

| | |
|---|---|
| `MIN(t)` | return minimum value of type *t* |
| `SIZE(t)` | return number of bytes required for a variable of type *t* |
| `HIGH(a)` | return upper bound of open array *a* |
| `ODD(i)` | return "*i* MOD 2 <> 0", type integer or char |
| `EVEN(i)` | return "*i* MOD 2 = 0", type integer or char |
| `CAP(c);` | return capital character, corresponding to *c* |

## General Procedures

| | |
|---|---|
| `NEW(p)` | allocate memory for a new data element, set *p* to address |
| `DISPOSE(p)` | deallocate memory for the element point to by *p*<br>pointer data type is `POINTER TO base_type`<br>nil element is `NIL` |
| `DEC(i)` | $i := i - 1$, type integer or character |
| `DEC(i,n)` | $i := i - n$, type integer or character |
| `INC(i)` | $i := i + 1$, type integer or character |
| `INC(i,n)` | $i := i + n$, type integer or character |
| `EXCL(s,n)` | exclude integer *n* from set *s* |
| `INCL(s,n)` | add integer *n* to set *s* |
| `HALT` | terminate program execution |

## Command Line Arguments

`PROCEDURE argc: CARDINAL;`
number of command line parameters

`PROCEDURE argv(index: CARDINAL; VAR arg: ARRAY OF CHAR);`
get command line parameters (range 1..*argc*)

## Mathematical Functions

The following functions have real arguments and return real values (scalar or vector).

| | |
|---|---|
| `pi` | constant $\pi = 3.14159265358979323 85$ |
| `sqrt(r)` | square root, $\sqrt{r}$ |
| `exp(r)` | exponent, $e^r$ |
| `ln(r)` | natural logarithm, $\ln(r)$ |
| `sin(r)` | sine |
| `arcsin(r)` | arcus sine |
| `cos(r)` | cosine |
| `arccos(r)` | arcus cosine |
| `tan(r)` | tangent |
| `arctan(r)` | arcus tangent, range $-\pi/2$ .. $+\pi/2$ |
| `arctan2(r1,r2)` | arcus tangent of $r_1 / r_2$ in range $-\pi$ .. $+\pi$ |

The following functions have real arguments and return integer values.

| | |
|---|---|
| `ceiling(r)` | round *r* to the next higher integer |
| `floor(r)` | round *r* to the next lower integer |
| `round(r)` | round *r* to the closest integer |

The following functions generate random numbers. For the parallel versions, a vector variable may be used in lieu of a configuration name `conf`.

| | |
|---|---|
| `RandomInt()` | generate a scalar integer random (`0..MAX(INTEGER)`) |
| `RandomCard()` | generate a scalar cardinal random (`0..MAX(CARDINAL)`) |
| `RandomReal()` | generate a scalar real random number (range `0..1`) |

| | |
|---|---|
| `RandomChar()` | generate a scalar character random value (`CHR(0..255)`) |
| `RandomBool()` | generate a scalar boolean random value (`TRUE` or `FALSE`) |
| `RandomInt(conf)` | generate a vector integer random (`0..MAX(INTEGER)`) |
| `RandomCard(conf)` | generate a vector cardinal random (`0..MAX(CARDINAL)`) |
| `RandomReal(conf)` | generate a vector real random number (range `0..1`) |
| `RandomChar(conf)` | generate a vector character random value (`CHR(0..255)`) |
| `RandomBool(conf)` | generate a vector boolean random value (`TRUE` or `FALSE`) |

### Input/Output Procedures

The following procedures may take either scalar or vector arguments. I/O operations do *not* generate run-time errors, but set the scalar boolean variable `Done` according to the success of the operation. All read operations besides `Read` set the scalar character variable `termCH`.

Vector data is printed in ascending `ID` order for all active PEs. Some rudimentary formatting is performed, when printing vector data. All integer, real, boolean, or string *vector* data (but not vector characters) will be separated by blanks when printed. A carriage return (new line) is inserted after the end of each dimension (e.g. this will print a two-dimensional vector in matrix format).

| | |
|---|---|
| `WriteLn;` | Start a new line (no arguments) |
| `Write(c);` | Write character *c* |
| `WriteString(s)` | Write string *s* |
| `WriteInt(i,l);` | Write integer *i* using *l* print spaces<br>(right adjusted: add blanks in front of number) |
| `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,l);` | Write boolean *b* using *l* print spaces<br>(left adjusted: add blanks after string) |
| `Read(c);` | Read character *c* |
| `ReadString(s)` | Read string *s* |
| `ReadInt(i);` | Read integer *i* |
| `ReadCard(c);` | Read cardinal *c* |
| `ReadReal(r);` | Read real *r* |
| `ReadBool(b);` | Read boolean *b* |

The following procedures take scalar arguments only.

| | |
|---|---|
| `OpenOutput(s);` | Open file with name *s* for writing<br>(if *s* is empty, then the user is prompted for a file name; following write operations write to file) |
| `CloseOutput;` | Close file, redirect output to *stdout* |
| `OpenInput(s);` | Open file with name *s* for reading<br>(if *s* is empty, then the user is prompted for a file name; following read operations read from file) |
| `CloseInput;` | Close file, redirect input back to *stdin* |

### Vector Functions

For the following functions and procedures, who take a configuration name as argument, a vector variable may be used in lieu of a configuration name. Argument `num` is a positive integer, `dims` is a vector array of integers.

| | |
|---|---|
| `ID(conf)` | returns a vector of identification numbers from 1 to `LEN(conf)` for configuration *conf* |
| `DIM(conf,num)` | returns a vector of position numbers according to the dimension declaration of *conf*, dimensions are numbered from *right* to *left* from 1 to `RANK(conf)` |
| `LEN(conf)` | returns the total number of PEs of a configuration |
| `LEN(conf,num)` | returns the size of a dimension |
| `RANK(conf)` | returns the number of dimensions |
| `UPPER(conf,num)` | returns the upper bound of a dimension |
| `LOWER(conf,num)` | returns the lower bound of a dimension |
| `DIM2ID(conf,dims)` | transforms the appropriate number of dim-values into the corresponding id-value, may be used with scalar or vector parameters |
| `MOVE.dir(val)` | data transfer in predefined direction *dir*, sender and receiver have to be active |
| `RECEIVE.dir(val)` | data transfer in predefined direction *dir*, only receiver has to be active |
| `REDUCE.func(val)` | reduces vector val to a scalar value, by applying *func* (predefined: `SUM`, `PRODUCT`, `MAX`, `MIN`, `AND`, `OR`, `FIRST`, `LAST`, or user function with two input parameters and a return value of identical type) |

### Vector Procedures

| | |
|---|---|
| `ID2DIM(conf,num,dims)` | transforms an id-value into an array of corresponding dim-values, may be used with scalar or vector parameters |
| `SEND.dir(val, var)` | data transfer in predefined direction *dir*, variable *var* receives expression *val*, only sender has to be active |

## A.3. Graphics Interface

### X Window Graphics

Module `Graphics` provides a convenient interface in Parallaxis for generating graphics, based on the X window system. Like read/write operations, graphics operations do not generate runtime errors, but set the variable `Done` according to the success of an operation. Type `CARDINAL` restricts `INTEGER` to positive numbers ($\geq 0$).

```
DEFINITION MODULE Graphics;
(* created: T. Braunl, Oct.94 *)

TYPE binary = BOOLEAN;
     gray   = [0..255];
     color  = RECORD
                 red, green, blue: gray
              END;
     window = CARDINAL;
```

```
CONST b_black      = FALSE;
      b_white      = TRUE;
      g_black      =   0;
      g_white      = 255;
      c_black      = color(  0,  0,  0);
      c_white      = color(255,255,255);
      c_red        = color(255,  0,  0);
      c_green      = color(  0,255,  0);
      c_blue       = color(  0,  0,255);

PROCEDURE OpenWindow(title: ARRAY OF CHAR; width,height: CARDINAL):
window;
PROCEDURE CloseWindow(window_num: window);
PROCEDURE SelectWindow(window_num: window);
   (* OpenWindow: Open a new window of specified title and size in *)
   (* pixels, return window number, activate new window            *)
   (* CloseWindow: close window; SelectWindow: activate window     *)

PROCEDURE GetScreenSize(VAR width,height: CARDINAL);
PROCEDURE GetWindowSize(VAR width,height: CARDINAL);
   (* Returns size in pixels of whole screen or active window, resp. *)

PROCEDURE SetArea (c: VECTOR OF color);
PROCEDURE SetgArea(c: VECTOR OF gray);
PROCEDURE SetbArea(c: VECTOR OF binary);
PROCEDURE SetAreaXYZ (c: VECTOR OF color;  x,y,zoom: INTEGER);
PROCEDURE SetgAreaXYZ(c: VECTOR OF gray;   x,y,zoom: INTEGER);
PROCEDURE SetbAreaXYZ(c: VECTOR OF binary; x,y,zoom: INTEGER);
   (* All active PEs write one pixel each, assuming a 2-dim. grid     *)
   (* configuration of appropriate size, image will be truncated or   *)
   (* left partly unchanged if PE grid size does not match image size *)
   (* origin is top left corner with coordinates 0, 0                 *)
   (* version "g" for gray data, version "b" for binary data          *)
   (* versions "XYZ" with offset and zoom (both pos. or neg.)         *)
   (* zoom range 1..256 determines pixel size, neg. zoom reduces size *)

PROCEDURE SetLine (c: VECTOR OF color;  line: CARDINAL);
PROCEDURE SetgLine(c: VECTOR OF gray;   line: CARDINAL);
PROCEDURE SetbLine(c: VECTOR OF binary; line: CARDINAL);
   (* print PE data along specified line, top line has no. 0 *)

PROCEDURE SetColumn (c: VECTOR OF color;  col: CARDINAL);
PROCEDURE SetgColumn(c: VECTOR OF gray;   col: CARDINAL);
PROCEDURE SetbColumn(c: VECTOR OF binary; lin: CARDINAL);
   (* print PE data along specified column, leftmost line has no. 0 *)

PROCEDURE SetColor (c: color);
PROCEDURE SetgColor(c: gray);
PROCEDURE SetbColor(c: binary);
          (* Specify color for subsequent pixel/line/draw commands *)
          (* "g" and "b" version for gray and binary data          *)
```

```
PROCEDURE SetPixel(x,y: CARDINAL);
PROCEDURE GetPixel(x,y: CARDINAL): color;
PROCEDURE Line(x1,y1, x2,y2: CARDINAL);
          (* scalar functions for writing or reading a single pixel *)
          (* scalar line: straight line from y1,x1 to y2,x2         *)

PROCEDURE DrawAt(x,y: CARDINAL);
   (* Set drawing position for subsequent drawing commands *)
   (* initialized with lower left character position (1,1) *)
   (* position will be updated by each drawing command     *)

PROCEDURE Draw      (c: CHAR);
PROCEDURE DrawString(s: ARRAY OF CHAR);
PROCEDURE DrawInt   (i,l: INTEGER);
PROCEDURE DrawCard  (c,l: CARDINAL);
PROCEDURE DrawReal  (r: REAL; l: CARDINAL);
PROCEDURE DrawFixPt (r: REAL; l,m: CARDINAL);
PROCEDURE DrawBool  (b: BOOLEAN; l:CARDINAL);
PROCEDURE DrawLn;
   (* Draw character c in window; Draw string s in window           *)
   (* Draw integer/cardinal i in window, using l print spaces       *)
   (* Draw real r in window, using l print spaces                   *)
   (* Draw boolean value b in window, using l print spaces left adj. *)
   (* DrawLn set drawing position to new line                       *)

END Graphics.
```

### Image File Transfer

Module `ImageIO` contains a number of operations for reading images from files and writing images to files (no X window interface is needed). The binary versions of the PPM (portable pixel map) file format are used for color, gray scale, and binary images.

```
DEFINITION MODULE ImageIO;
(* created: T. Braunl, Oct.94                           *)
(* ********************************************** *)
(* read_c_image   read color image in  ppm P6 format *)
(*                into a vector variable             *)
(* read_g_image   read gray image in   ppm P5 format *)
(* read_b_image   read binary image in ppm P4 format *)
(*                                                   *)
(* write_c_image  write volor image  in P6 format    *)
(* write_g_image  write gray image   in P5 format    *)
(* write_b_image  write binary image in P4 format    *)
(*                                                   *)
(* PE-ID 1 (origin) is top left                      *)
(* ********************************************** *)

CONST b_black      = FALSE;
      b_white      =  TRUE;
      g_black      =     0;
      g_white      =   255;
```

```
      c_black      = color(  0,  0,  0);
      c_white      = color(255,255,255);
      c_red        = color(255,  0,  0);
      c_green      = color(  0,255,  0);
      c_blue       = color(  0,  0,255);

TYPE binary = BOOLEAN;
     gray   = [0..255];
     color  = RECORD
                red, green, blue: gray
              END;

PROCEDURE read_c_image (VAR im: VECTOR OF color;filename: ARRAY OF CHAR;
                        VAR width,height: CARDINAL);
PROCEDURE read_g_image (VAR im: VECTOR OF gray; filename: ARRAY OF CHAR;
                        VAR width,height: CARDINAL);
PROCEDURE read_b_image (VAR im: VECTOR OF binary;filename:ARRAY OF CHAR;
                        VAR width,height: CARDINAL);

PROCEDURE write_c_image (im: VECTOR OF color;  filename: ARRAY OF CHAR;
                         width,height: CARDINAL);
PROCEDURE write_g_image (im: VECTOR OF gray;   filename: ARRAY OF CHAR;
                         width,height: CARDINAL);
PROCEDURE write_b_image (im: VECTOR OF binary; filename: ARRAY OF CHAR;
                         width,height: CARDINAL);

END ImageIO.
```

## A.4. Parallaxis-III Syntax

Parallaxis-III Syntax is specified in EBNF (Extended Backus-Naur-Form).

```
1   CompilationUnit         =  ProgramModule
                               | DefinitionModule
                               | ImplementationModule
                               | ForeignModule .
2   ProgramModule           =  MODULE Ident ';' { Import } Block Ident '.' .
3   DefinitionModule        =  DEFINITION MODULE Ident ';'
                               { Import } [ Export ] { Definition } END Ident '.' .
4   ImplementationModule    =  IMPLEMENTATION MODULE Ident ';'
                               { Import } Block Ident '.' .
5   ForeignModule           =  FOREIGN MODULE Ident ';'
                               { Import } [ Export ] { Definition } END Ident '.' .

6   Definition              =  CONFIGURATION {ConfigDeclaration ';'}'
                               | CONNECTION { ConnectionDeclaration ';'}
                               | CONST {ConstantDeclaration ';'}
                               | TYPE {Ident [ '=' GeneralType ] ';'}
                               | VAR {VariableDeclaration ';'}
                               | ProcedureHeading ';' .
7   Import                  =  [ FROM Ident ] IMPORT IdentList ';' .
8   Export                  =  EXPORT [ QUALIFIED ] IdentList ';' .
9   Block                   =  {Declaration } [ BEGIN StatementSequence ] END .
10  Declaration             =  CONFIGURATION {ConfigDeclaration ';'}
                               | CONNECTION {ConnectionDeclaration ';'}
                               | CONST {ConstantDeclaration ';'}
                               | TYPE {TypeDeclaration ';'}
                               | VAR {VariableDeclaration ';'}
                               | ProcedureDeclaration ';' .

11  ConfigDeclarartion      =  Config {';' Config } .
12  Config                  =  [ Ident [ '=' Qualident ] ConfigRange { ',' ConfigRange } ] .
13  ConfigRange             =  '[' ConstExpression '..' ConstExpression ']‘
                               | '[' '*' ']' .
14  ConnectionDeclaration   =  TransferFunction
                               | FOR Ident ':=' Expression TO Expression DO
                               TransferFunction {';' TransferFunction } END .
15  TransferFunction        =  [ Direction ':' Qualident '[' Source {',' Source }']'
                               ( '–>' Dest1 {',' Dest1} | '<–>' Dest2 {',' Dest2} ) ]
16  Direction               =  Ident [ '[' Source ']' ] .
17  Source                  =  Ident
                               | Integer
                               | String
                               | '*' .
18  Dest1                   =  [ Discriminant ] Qualident '[' DestExprList ']' .
19  Dest2                   =  [ Discriminant ] Qualident '[' DestExprList ']' :' Ident
                               [ '[' Expression ']' ] .
20  DestExprList            =  DestExpr {',' DestExpr } .
21  DestExpr                =  Expression
                               | Expression '..' Expression
                               | '*' .
22  Discriminant            =  '{' Expression '}' .

23  ConstantDeclaration     =  Ident '=' ConstExpression .
24  ConstExpression         =  Expression .
25  TypeDeclaration         =  Ident '=' GeneralType .
26  GenaralType             =  SimpleType
                               | GeneralArrayType
                               | GeneralRecordType
                               | SetType
```

```
                                     | PointerType
                                     | ProcedureType
                                     | ( VECTOR | Qualident )  [ Qualident ]  OF  ScalarType .
27  ScalarType                  =    SimpleType
                                     | ScalarArrayType
                                     | ScalarRecordType
                                     | SetType
                                     | PointerType
                                     | ProcedureType .
28  GeneralArrayType            =    ARRAY  SimpleType  { ','  SimpleType } OF GeneralType .
29  GeneralRecordType           =    RECORD  FieldListSequence1  END  .
30  FieldListSequence1          =    [ FieldList1  { ';'  FieldList1 }  ] .
31  FieldList1                  =    IdentList ':'  GeneralType
                                     | CASE  [ Ident ] ':'  Qualident  OF  Variant { '|' Variant}
                                     [ ELSE  FieldListSequence2 ]  END  .
32  Variant                     =    [ CaseLabelList ':'  FieldListSequence2 ] .
33  CaseLabelList               =    CaseLabels  {  ‘,'  CaseLabels  }   .
34  CaseLabels                  =    ConstExpression [ '..'  ConstExpression ] .

35  ScalarArrayType             =    ARRAY  SimpleType  { ','  SimpleType } OF ScalarType .
36  ScalarRecordType            =    RECORD  FieldListSequence2  END  .
37  FieldListSequence2          =    [ FieldList2  {';'  FieldList2 }  ] .
38  FieldList2                  =    IdentList ':'  ScalarType
                                     | CASE  [ Ident ] ':'  Qualident  OF  Variant { '|' Variant }
                                     [ ELSE  FieldListSequence2 ]  END  .

39  SimpleType                  =    Qualident
                                     | Enumeration
                                     | SubrangeType .
40  Enumeration                 =    '(' IdentList ')' .
41  SubrangeType                =    [ Qualident ]  '[‘ ConstExpression '..'  ConstExpression ']' .
42  SetType                     =    SET  OF  SimpleType .
43  PointerType                 =    POINTER  TO  GeneralType .
44  ProcedureType               =    PROCEDURE  [ FormalTypeList ] .
45  FormalTypeList              =    '(' [ [ VAR ] FormalType { ';'  [ VAR ]  FormalType  } ] ')'
                                     [ ':'  [ ( VECTOR | Qualident ) OF ]  Qualident .
46  FormalType                  =    [ ( VECTOR | Qualident )  OF ]  [ ARRAY  OF ] Qualident .
47  VariableDeclaration         =    IdentList ':'  GeneralType .

48  ProcedureDeclaration        =    ProcedureHeading ';'  Block Ident .
49  ProcedureHeading            =    PROCEDURE  Ident [ FormalParameters ] .
50  FormalParameters            =    '(' [ Parameter  { ';'  Parameter }  ] ')' [ ':'  Qualident ] .
51  Parameter                   =    [ VAR ] IdentList ':'  FormalType .

52  StatementSequence           =    Statement  {';'  Statement  } .
53  Statement                   =    [ Assignment
                                     | ProcedureCall
                                     | IfStatement
                                     | CaseStatement
                                     | WhileStatement
                                     | RepeatStatemen
                                     | LoopStatement
                                     | ForStatement
                                     | WithStatement
                                     | EXIT
                                     | RETURN  [ Expression ]
                                     | AllStatement
                                     | SendStatement
                                     | LoadStatement
                                     | StoreStatement ] .
54  Assignment                  =    Designator ':='  Expression .
55  ExprList                    =    Expression  { ','  Expression }  .
```

```
56  Expression          =  SimpleExpression  { RelationOperator  SimpleExpression }.
57  RelationOperator    =  '=' | '#' | '<>' | '<' | '<=' | '>' | '>=' | IN .
58  SimpleExpression    =  [ '+' | '–' ] Term  { AddOperator Term }  .
59  AddOperator         =  '+' | '–' | OR  .
60  Term                =  Power  { MulOperator Power }  .
61  MulOperator         =  '*' | '/' | DIV | MOD | AND | '&' .
62  Power               =  Factor  { '**' Factor }  .
63  Factor              =  Number
                           | String
                           | Set
                           | Designator [ ActualParameters ]
                           | MoveFunction
                           | ReceiveFunction
                           | ReduceFunction
                           | ArrayInitializer
                           | RecordInitializer
                           | '(' Expression ')'
                           | NOT  Factor
                           | '~' Factor .
64  Set                 =  [ Qualident ] '{' [ Element  { ','  Element  } ] '}' .
65  Element             =  ConstExpression [ '..'  ConstExpression ] .
66  ActualParameters    =  '(' [ ExprList ] ')' .

67  MoveFunction        =  MOVE '.' DirSpecifier '(' Expression ')' .
68  DirSpecifier        =  Ident [ '[' Expression ']' ]  [ ':#' ReductIdent ]
                           | '<<' Expression '>>'  [ ':#' ReductIdent ]
                           |  '<:' Dimension { ',' Dimension } ':>'  [ ':#' ReductIdent ].
69  SendSpecifier       =  Ident [ '[' Expression ']' ]  [ ':' StepSpecifier ]  [ ':#' ReductIdent ]
                           | '<<' Expression '>>'  [ ':#' ReductIdent ]
                           |  '<:' Dimension { ',' Dimension } ':>'  [ ':#' ReductIdent ] .
70  StepSpecifier       =  '(' Expression ')' | Qualident | Integer .
71  ReductIdent         =  AND | OR | Qualident .

72  ReceiveFunction     =  RECEIVE  '.'  DirSpecifier '(' Expression ')' .
73  ReduceFunction      =  REDUCE  '.'  ReductIdent '(' Expression ')' .

74  ArrayInitializer    =  Qualident '(' ExprList ')' .
75  RecordInitializer   =  Qualident '(' ExprList ')' .
76  ProcedureCall       =  Designator [ ActualParameters ] .

77  IfStatement         =  IF  Expression  THEN  StatementSequence
                           { ELSIF  Expression  THEN  StatementSequence}
                           [ ELSE  StatementSequence ]  END  .
78  CaseStatement       =  CASE  Expression  OF  Case  { '|' Case }
                           [ ELSE  StatementSequence ]  END  .
79  Case                =  [ CaseLabelList ':'  StatementSequence ] .

80  WhileStatement      =  WHILE  Expression  DO  StatementSequence  END  .
81  RepeatStatement     =  REPEAT  StatementSequence  UNTIL  Expression .
82  ForStatement        =  FOR  Ident ':='  Expression  TO  Expression
                           [ BY  ConstExpression ]  DO  StatementSequence  END  .
83  LoopStatement       =  LOOP  [ OF Qualident DO ]  StatementSequence  END  .

84  WithStatement       =  WITH  Designator  DO  StatementSequence  END  .
85  AllStatement        =  ALL  Qualident  DO  StatementSequence  END  .
86  SendStatement       =  SEND  '.'  SendSpecifier  '(' Expression ',' Designator ')' .

87  LoadStatement       =  LOAD  '(' Designator  ','  Designator [ ','  Designator ] ')' .
88  StoreStatement      =  STORE  '(' Designator  ','  Designator [ ','  Designator ] ')' .
89  Designator          =  Qualident  { << Expression >>  |  <: ExprList :>
                           |  '.'  Ident |  '['  ExprList ']' |  '^'  } .
90  Dimension           =  Expression  | '*'  | '#' ReductIdent .
```

| | | | |
|---|---|---|---|
| 91 | Qualident | = | Ident [ '.' Ident ] . |
| 92 | IdentList | = | Ident { ',' Ident } . |
| 93 | Ident | = | Letter { Letter \| Digit } . |
| 94 | Letter | = | ( 'A' .. 'Z' \| 'a' .. 'z' \| '_' ) . |
| 95 | String | = | ' ' ' { Character } ' ' ' \| ' " ' { Character } ' " ' . |
| 96 | Number | = | Integer \| Real . |
| 97 | Integer | = | Digit { Digit } [ 'D' ] |
| | | | \| OctalDigit { OctalDigit } ( 'B' \| 'C' ) |
| | | | \| Digit { HexDigit } 'H' . |
| 98 | Real | = | Digit { Digit } '.‘ { Digit } [ ScaleFactor ] . |
| 99 | ScaleFactor | = | 'E' [ '+' \| '–' ] Digit { Digit } . |
| 100 | HexDigit | = | Digit \| 'A' \| 'B' \| 'C' \| 'D' \| 'E' \| 'F' . |
| 101 | Digit | = | OctalDigit \| '8' \| '9' . |
| 102 | OctalDigit | = | '0' \| '1' \| '2' \| '3' \| '4' \| '5' \| '6' \| '7' . |