OPENINPUT name

Redirect all following input from the file with the given name, which may be a string, a char constant or a char variable (string terminated by `termS`). If *name* is the empty string, the name is read from the terminal with the prompt "`in> `". The file identified by *name* is opened and substitutes the standard input stream. `Done` is `TRUE` if the file could be opened, `FALSE` otherwise.

CLOSEINPUT

If input was redirected from a file, this command closes the file and switches back to standard input.

OPENOUTPUT name

In analogy to `OPENINPUT`, this procedure redirects output to the file identified by name. If the name is read from the terminal, the prompt is "`out> `". `Done` is `TRUE` if the file could be opened, `FALSE` otherwise.

CLOSEOUTPUT

If output was redirected to a file, this command closes the file and switches back to standard output.

ERROR string

This procedure writes an error message containing the constant *string* to the console and stops program execution.

Debug Procedures

DEBUG vardesc string AS declaration

vardesc and *declaration* define a block of variables that will be printed together with the supplied *string*. The value of each variable is written on a separate line. Character arrays declared "`AS Cnum`" are considered a string. Strings appear in the output enclosed in quotes and may extend over several lines. For each parenthesis-level output is indented by two characters. Every repetition of a parenthesized list of sub-declarations is followed by '|' if that list declares more than one variable. Variants in a union are terminated by a comma, union ends are marked with a semicolon. Un-initialized values are shown in the form '`x : ??`', where X is one of the type characters B, C, I or R. In strings, special representations of characters like `termS` or `CHR(255)` are enclosed in '<' and '>'. If *vardesc* is a vector, the blocks of as many PEs as possible are written side by side.

TRACE vardesc string AS declaration

The arguments are identical to the `DEBUG` procedure, but *vardesc* may not be an indirect variable with vector index. The block defined by *vardesc* and *declaration* will be written to the output stream (together with the supplied string) when the `TRACE` procedure is executed, and every time one of the variables in the block is assigned a new value. The output format is identical to the `DEBUG` command, but shows the command causing the value change in addition.

NOTRACE [vardesc]

Without argument, it disables tracing on all blocks; with an argument, all blocks beginning with *vardesc* will no longer be traced. The same restriction for indirect variables applies as for the `TRACE` procedure.

7.7 Graphics Interface

The following commands are integrated in the language PARZ. All commands set the predefined variable `Done` according to the success of the operation.

s_vardesc ":=" OPENW s_VarConst_x s_VarConst_y

s_VarConst_x and *s_VarConst_y* are two values in the range (0.0 ... 1.0), denoting width and height of the graphical output window, relative to the maximum screensize. A unique window number is assigned to the left hand side argument.

s_vardesc ":=" OPENABSW s_VarConst_x s_VarConst_y

s_VarConst_x and *s_VarConst_y* are two integer values, denoting width and height of the graphical output window in absolute pixel values. A unique window number is assigned to the left hand side argument.

SELECTW s_VarConst

The window with number *s_VarConst* is activated for graphical output.

WSIZE s_vardesc_x s_vardesc_y

Width and height in pixels of the active window are assigned to *s_vardesc_x* and *s_vardesc_y*.

CLOSEW s_VarConst

The window with number *s_VarConst* is closed.

SETCOLOR vardesc

vardesc is the first of three continuous variables, containing the values of the new actual drawing color. *vardesc* may be a scalar or a vector value (setting an individual color for each PE).

SETPIXEL VarConst_x VarConst_y

In case of scalar arguments, the pixel with coordinates (*VarConst_x*, *VarConst_y*) is set to the actual drawing color. In case of vector arguments, each PE sets a pixel at the supplied coordinates, either in the same or an individual color, depending on the previous call of SETCOLOR.

s_vardesc ":=" GETPIXEL s_VarConst_x s_VarConst_y

The left hand side argument is the first of three continuous INTEGER variables receiving the color values (red, green, blue) of the specified pixels.

MOVETO s_VarConst_x s_VarConst_y

Defines the start position for the next call of LINETO.

LINETO s_VarConst_x s_VarConst_y

Draws a line in the actual drawing color from the last position reached with MOVETO or LINETO to position (*s_VarConst_x*, *s_VarConst_y*).

DRAW s_VarConst1 [s_VarConst2 [s_VarConst3]]

Writes the scalar value of *s_VarConst1* to the selected window at the actual position. With only one argument, a standard format is used for values of each type (boolean, char, integer, real, string). With two arguments, *s_VarConst2* is the minimal output length for integer or real values. If *s_VarConst1* is a character variable, it is the first of a character array containing a string of maximal length *s_VarConst2*. Three arguments are used to write real values in a fix-point notation. Here, *s_VarConst2* is the minimal output width and *s_VarConst3* is the fraction width.

8 Compiling Executables

Two compilers have been implemented. Compiler `pz2c` takes a PARZ file as input and generates portable Kernighan&Ritchie C programs to build faster stand-alone applications. The second compiler, `pz2mpl`, takes a PARZ file as input and generates a parallel MPL program for the massively parallel MasPar MP-1 system. A compiler for the Connection Machine 2 is just being completed.

Both compilers should only be used after testing a Parallaxis program with the PARZ simulator / debugger, because no run-time checks are generated.

The pseudo-comment $R {proc_ident} in a previously translated Parallaxis program may be used for both compilers to generate run time measurements. The CPU-times for all given procedures are measured while executing the program and are printed into a file with name:
    <executable_name>`.timing` .

8.1 PARZ to C

The compiler is started by typing

```
     pz2c {options} inFilename [-o exeFilename]
```

Compiling PARZ programs into C programs gives two advantages:

- *compiled* Parallaxis/PARZ programs are faster than *interpreted* ones
- *compiled* Parallaxis/PARZ programs are easily portable, since almost every computer system comes with a C compiler.

Options start with a '-' sign. All characters after the '-' sign up to the next white space are treated as options. A compiler call may contain more than one option.

The options for both compilers are:

d	use double precision for data type REAL
enum	set the maximum number of reported error messages
g	activate global timing
m	call the C/MPL compiler for existing code
O	invoke the optimizer (capital O) of the C/MPL compiler
p	first run the Parallaxis compiler
r	remove generated C/MPL source code after compiling it
R	activate recording mode
t	generate trace information for all PARZ labels (only for debugging)
u	hand option "-u" over to the Parallaxis compiler
v	verbose mode, give information about the compiling
wnum	set the maximum number of reported warnings
x	create an identifier cross-reference file. This file will have the extension `.ref`
z	don't call the C/MPL compiler after generating the C/MPL source code

o <executable>   set the name of the generated executable file

The compiler takes files ending with `.z` as input files. If the filename entered has not this extension, the compiler adds `.z` and tries to open this file.

If option `-p` is provided or the inFilename has the extension `.p`, the Parallaxis compiler is invoked with the options `-dmnrx`, i.e. DEBUG and TRACE commands are ignored, no PARZ code for operand-checking in mathematical operations, range-check, or `NIL`-pointer de-referentiation is generated.

The name of the generated C program is built by removing the standard extension from the inFilename and adding the extension `.c` to it. The name of the generated executable file is normally built by removing the standard extension from the inFilename. If option `-o exeFilename` is provided, this filename is used for the executable program.

Note, that no DEBUG or TRACE information is supported, i.e. DEBUG and TRACE statements in the input file are skipped and a warning is reported.

To create a stand-alone executable file, two files are required:

• `pz2c.h`	the C include file
• `libpz2cs.a`	the linking library for single precision on a UNIX system;
`libpz2cd.a`	the linking library for double precision on a UNIX system (See option "-d" for arithmetic precision of data type "real".)

If you cannot copy these files in the standard search paths for include files and libraries, set the environment variables `PZ2C_HEADER` and `PZ2C_LIB` to the directories, which contain these files. E.g., if the file `libpz2c.a` resides in the directory `/home/smith/lib`, the variable `PZ2C_LIB` must have the value `'/home/smith/lib'`.

8.2 PARZ to MPL

The compiler is started by typing

```
pz2mpl  {options} inFilename [-o exeFilename]
pz2mpls {options} inFilename [-o exeFilename]
```

`pz2mpl` — *for programs with much data*
`pz2mpls` — *for programs with few data*

These compilers generate MPL programs from PARZ intermediate code. MPL (MasPar Parallel Application Language) is a company-specific data parallel extension of C. Options and behaviour of this compiler are identical to the PARZ-to-C compiler, refer to section 8.1 for details.

The name extension for generated MPL programs is `.m` . Depending on whether compiling with `pz2mpl` ("large model") or `pz2mpls` ("small model"), the include and library file names are as follows:

Include Files:

`pz2mpl-fe.h, pz2mpl-dpu.h`	large model MPL
`pz2mpls.h`	small model MPL

Libraries:

`libpz2mpl-fes.a,`	`libpz2mpl-dpus.a`	large model, single prec.
`libpz2mplss.a,`	`libgraphs.a`	small model, single prec.
`libpz2mpl-fed.a,`	`libpz2mpl-dpud.a`	large model, double prec
`libpz2mplsd.a,`	`libgraphd.a`	small model, double prec.

The large model always requires linking of two libraries, one for the front end (FE) and one for the data parallel unit (DPU). The small model requires one or two libraries, depending on whether the Parallaxis application program contains calls of graphics routines or not. For the large model, this additional graphics library is already included in the pz2mpl-fe library.

As for the PARZ-to-C compiler, depending on the selection of single or double precision arithmetic for numbers of type "real" (see option "-d"), the appropriate libraries have to be used.

For MPL, the environment variables are called

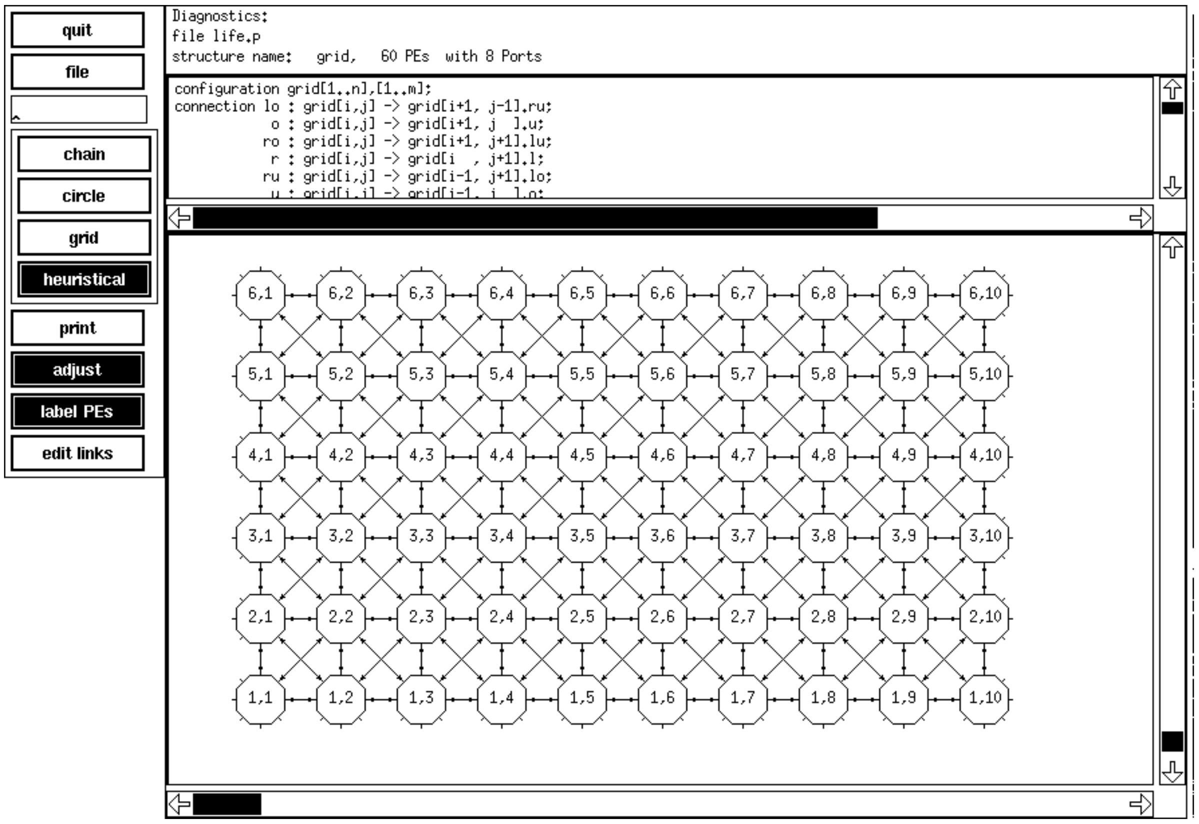
`PZ2MPL_HEADER`	(`PZ2MPLS_HEADER`, resp.)	and
`PZ2MPL_LIB`	(`PZ2MPLS_LIB`, resp.).	

Compiler `pz2mpl` generates a C program for the front-end and an MPL program for the MasPar back-end. All scalar data resides in the front-end. This allows large virtual data space, but requires a lot of overhead for data exchange between front-end system and MasPar. For programs that get by with the MasPar's very limited 114 KB scalar data memory, the compiler `pz2mpls` generates much faster code, for all scalar data resides in the MasPar; no data exchange with the front-end is required.

9 Tools

There are a number of tools in development for Parallaxis. Although they are not yet ready for public domain distribution with version 2 of the language, we nevertheless feel like introducing them here.

The Visualizer is a tool for X-window systems, to generate a graphics representation of the network topology of a Parallaxis program. Processing elements are displayed as polygons (depending on the number of links) and may be arranged as a list, ring, or two-dimensional grid. The positions of the PE-links may be edited with a graphics tool. A typical example is shown in figure 9.1 . This tool has been implemented by Schulze and Christ and is documented in [Schulze Christ 90].



```
Diagnostics:
file life.p
structure name:   grid,   60 PEs  with 8 Ports
```

```
configuration grid[1..n],[1..m];
connection lo : grid[i,j] -> grid[i+1, j-1].ru;
            o : grid[i,j] -> grid[i+1, j  ].u;
           ro : grid[i,j] -> grid[i+1, j+1].lu;
            r : grid[i,j] -> grid[i  , j+1].l;
           ru : grid[i,j] -> grid[i-1, j+1].lo;
            u : grid[i,j] -> grid[i-1, j  ].o;
```

Figure 9.1: visualization of a grid topology

Another "tool" is the timing diagram. Actually it's a rather simple program, which extracts information from the recording file after a simulation run. The data is somewhat adjusted by coefficients taking care of different execution times of arithmetic pseudo-assembly commands versus parallel data exchange, load / store, and reduce commands. The resulting list of activation values gives at least a rough impression of the load versus time behaviour of a data parallel program. The data is being fed into a spreadsheet program and gives characteristic curves, revealing program parts with good load values and program parts that may be subject to improvement. Figure 9.2 shows the diagram of the sample sorting program from section 10.4 .

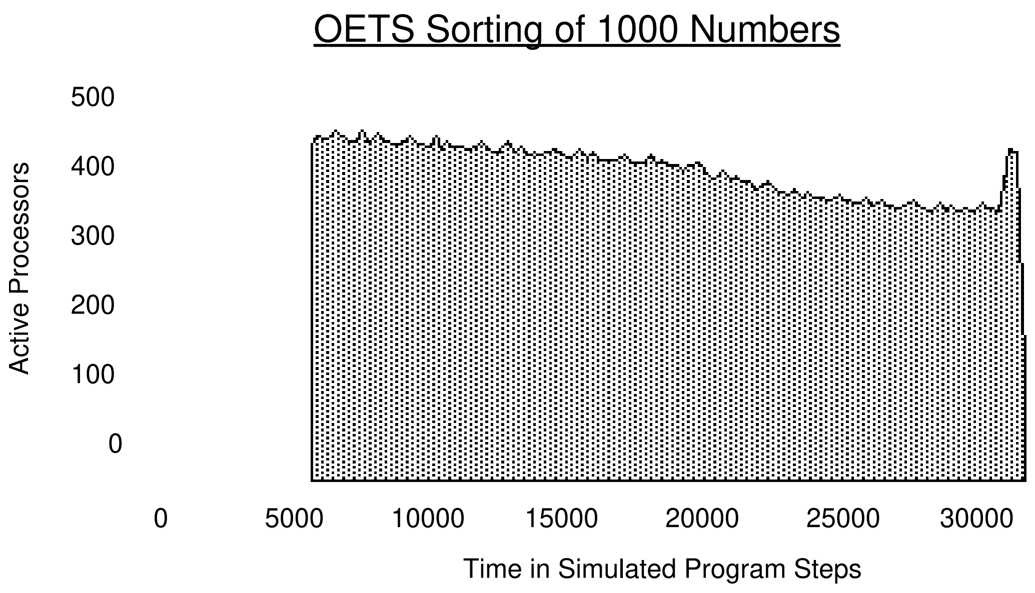


Figure 9.2: timing diagram for sorting program

10 Sample Programs

Now, we'd like to show a number of data parallel sample programs. The first will be discussed on all levels: high level language Parallaxis, intermediate language PARZ, and execution recording. Some example programs have been abbreviated for clarity. You will find the full source code of all example programs presented in the examples subdirectory of our ftp-server or on the supplied floppy disk.

10.1 Maximum Search

The first problem is to find the largest element from a set of n^2 elements. For demonstration purposes we **do not** use the reduction operation which can solve this problem with a single call in time $O(\log n^2)$. With n^2 processors and disregarding I/O, we complete this task in time O(n), more accurately in time 2*(n-2).

In the given two-dimensional processor arrangement, first the largest elements are being pushed from right to left, column by column, each PE selecting the larger of two elements. After n-1 steps, the leftmost column contains the largest elements. In the second step, all PEs push the largest value from top to bottom, until after another n-1 steps the bottom-leftmost element contains the largest element of the whole matrix. This PE is identified by position [0,0] in Parallaxis (`dim1` and `dim2`) and id-number 1 in PARZ (`ID`).

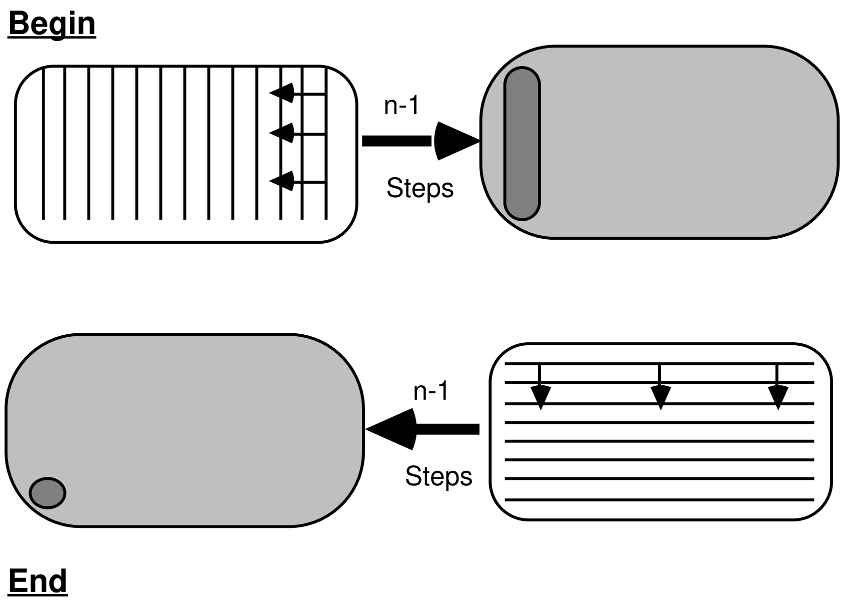
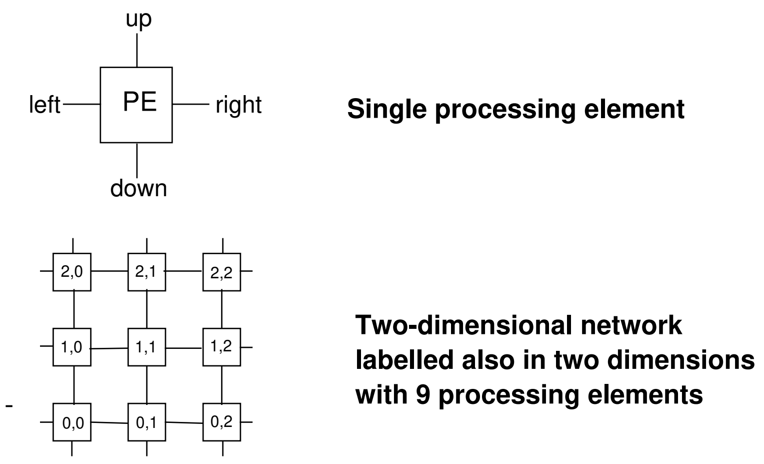


Figure 10.1: algorithm sketch

The following Parallaxis program implements this algorithm. To show all steps on a low level, we only use a 3×3 matrix.



up
left
PE
right
down
Single processing element
2,0
2,1
2,2
1,0
1,1
1,2
0,0
0,1
0,2
Two-dimensional network labelled also in two dimensions with 9 processing elements

10.2: Processing Elements as seen from Parallaxis

Program 1: Find largest element (Parallaxis)

```
SYSTEM FindMax;
CONST   size = 3;
CONFIGURATION  grid [size],[size];
CONNECTION   right:  grid[i,j]  ->  grid[i,j+1].left;
             left :  grid[i,j]  ->  grid[i,j-1].right;
             up   :  grid[i,j]  ->  grid[i+1,j].down;
             down :  grid[i,j]  ->  grid[i-1,j].up;

SCALAR  i      : integer;  (* variable for host *)
VECTOR  value,
        buffer: integer;   (* variables for each PE *)

BEGIN
  (* Initialize data: each PE gets its id_no as value *)
  PARALLEL     (* start parallel execution for all PEs *)
    value := id_no;
  ENDPARALLEL; (* stop parallel execution *)

  (* search maximum from rigth to left *)
  for i:=1 to size-1 do
    PARALLEL
      buffer := value;
      propagate.left(buffer);
      if buffer > value then value := buffer end
    ENDPARALLEL
  end;

  (* search maximum from top to bottom *)
  for i:=1 to size-1 do
    PARALLEL
      buffer := value;
      propagate.down(buffer);
      if buffer > value then value := buffer end
```

```
       ENDPARALLEL
     end;

     (* the largest value is now in PE (0,0) *)
     store [0],[0] (value, i);
     WriteInt(i,5)
   END FindMax.
```

This program is now being translated into the intermediate language PARZ. This is normally done by the compiler, but here we did it manually for clarity.

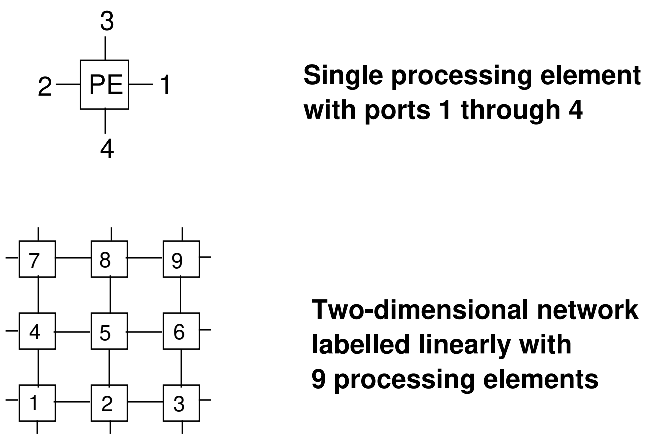


Figure 10.3: Processing Elements as seen from PARZ

As you can see from fig. 10.2 and fig. 10.3, the parallel system has a much simpler representation in PARZ than it does in Parallaxis. PEs as well as ports are just labelled from 1 to the number of PEs / ports.

Program 2: Find largest element (PARZ)

```
     START
     9 PE
     4 PORTS
     SCALAR i 1
     VECTOR i 3
         vi0:3 := id + 1;             address of right neighbor
         connect 1 to 2 at vi0:3;       establish connection
         vi0:3 := id - 1;               left neighbor
         connect 2 to 1 at vi0:3;

         vi0:3 := id + 3;               up neighbor
         connect 3 to 4 at vi0:3;
         vi0:3 := id - 3;               down neighbor
         connect 4 to 3 at vi0:3;

         vi0:1 := id;                  initialize with PE-id_no
         si0:1 := 1;                  loop counter right to left
```

```
          while si0:1 < 3 call 1;
          si0:1 := 1;                     loop counter up to down
          while si0:1 < 3 call 2;
          store vi0:1 to si0:1 PE 1;     get result from PE No. 1
          write si0:1;
          halt;

       1: proc 1;
            vi0:2 := vi0:1;
            propagate vi0:2 out 2 in 1;  propagate values left
            if vi0:2 > vi0:1 call 3;     select larger value
            si0:1 := si0:1 + 1;          increment counter
          return;

       2: proc 1;
            vi0:2 := vi0:1;
           propagate vi0:2 out 4 in 3;  propagate values downward
            if vi0:2 > vi0:1 call 3;     select larger value
            si0:1 := si0:1 + 1;          increment counter
          return;

       3: proc 2;
            vi0:1 := vi0:2;
          return;

          end;
      STOP
```

The PARZ simulator optionally produces an execution recording. There are different recording levels possible as discussed in the simulator description.

```
mode : 2

program : test.z

      1 :       VI0:3 := ID + 1;      address of right neighbor
          111111111 ;
     2 :       CONNECT 1 TO 2 AT VI0:3;     establish connection
          111111111 ;
      3 :       VI0:3 := ID - 1;                left neighbor
          111111111 ;
      4 :       CONNECT 2 TO 1 AT VI0:3;
          111111111 ;
      5 :       VI0:3 := ID + 3;                up neighbor
          111111111 ;
      6 :       CONNECT 3 TO 4 AT VI0:3;
          111111111 ;
      7 :       VI0:3 := ID - 3;                down neighbor
          111111111 ;
```

```
  8 :       CONNECT 4 TO 3 AT VI0:3;
      111111111 ;
  9 :       VI0:1 := ID;           initialize with PE-id_no
      111111111 ;
 10 :       SI0:1 := 1;            loop counter right to left
 11 :       WHILE SI0:1 < 3 CALL 1;
 12 : 1    : PROC 1;
 13 :       VI0:2 := VI0:1;
      111111111 ;
 14 :       PROPAGATE VI0:2 OUT 2 IN 1; propagate values left
      111111111 ;
 15 :       IF VI0:2 > VI0:1 CALL 3;      select larger value
      111111111 ;
 16 : 3    : PROC 2;
 17 :       VI0:1 := VI0:2;
      111111110 ;
 18 :       RETURN;
      111111110 ;
 19 :       SI0:1 := SI0:1 + 1;           increment counter
 20 :       RETURN;
 21 :       WHILE SI0:1 < 3 CALL 1;
 22 : 1    : PROC 1;
 23 :       VI0:2 := VI0:1;
      111111111 ;
 24 :       PROPAGATE VI0:2 OUT 2 IN 1;   propagate values left
      111111111 ;
 25 :       IF VI0:2 > VI0:1 CALL 3;      select larger value
      111111111 ;
 26 : 3    : PROC 2;
 27 :       VI0:1 := VI0:2;
      111111100 ;
 28 :       RETURN;
      111111100 ;
 29 :       SI0:1 := SI0:1 + 1;           increment counter
 30 :       RETURN;
 31 :       WHILE SI0:1 < 3 CALL 1;
 32 :       SI0:1 := 1;                loop counter up to down
 33 :       WHILE SI0:1 < 3 CALL 2;
 34 : 2    : PROC 1;
 35 :       VI0:2 := VI0:1;
      111111111 ;
 36 :       PROPAGATE VI0:2 OUT 4 IN 3;  propagate values down
      111111111 ;
 37 :       IF VI0:2 > VI0:1 CALL 3;      select larger value
      111111111 ;
 38 : 3    : PROC 2;
 39 :       VI0:1 := VI0:2;
      111111000 ;
 40 :       RETURN;
```

```
        111111000 ;
     41 :        SI0:1 := SI0:1 + 1;           increment counter
     42 :        RETURN;
     43 :        WHILE SI0:1 < 3 CALL 2;
     44 : 2    : PROC 1;
     45 :        VI0:2 := VI0:1;
        111111111 ;
     46 :        PROPAGATE VI0:2 OUT 4 IN 3;  propagate values down
        111111111 ;
     47 :        IF VI0:2 > VI0:1 CALL 3;      select larger value
        111111111 ;
     48 : 3    : PROC 2;
     49 :        VI0:1 := VI0:2;
        111000000 ;
     50 :        RETURN;
        111000000 ;
     51 :        SI0:1 := SI0:1 + 1;           increment counter
     52 :        RETURN;
     53 :        WHILE SI0:1 < 3 CALL 2;
     54 :        STORE VI0:1 TO SI0:1 PE 1;    get result from PE 1
        100000000 ;
     55 :        WRITE SI0:1;
output: 9
     56 :        HALT;
```

10.2 Image Rotation

The problem is to turn a raster picture on a two-dimensional grid about an angle of 90^o clockwise. One solution is to divide the picture in four quadrants, exchange their data cyclically and iterate the procedure at half grain size until the pixel-level is reached. Each quadrant may be computed independently of the other three quadrants, so this task can be solved in parallel.

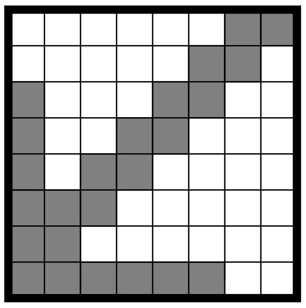
This algorithm is described for Smalltalk by Goldberg and Robsen in "Smalltalk 80, The Language and its Implementation".

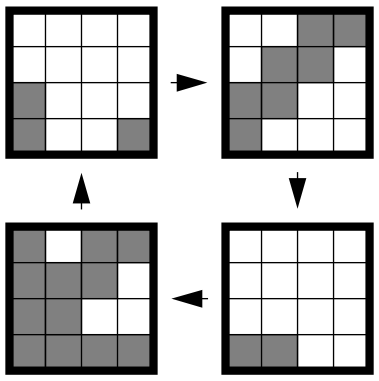
Programm 3: Image Rotation

```
SYSTEM  image_rot;
CONST   m_size = 1024;
CONFIGURATION  Pic [m_size],[m_size];
CONNECTION     right :  Pic [i, j]  <->  Pic [i, j+1].left;
               up    :  Pic [i, j]  <->  Pic [i+1, j].down;


VECTOR  color, buffer, b2: integer;


PROCEDURE rotate (SCALAR pic_size: integer);
(* assumtion: pic_size = 2^k *)
SCALAR size2: integer;
VECTOR x,y  : integer;
```

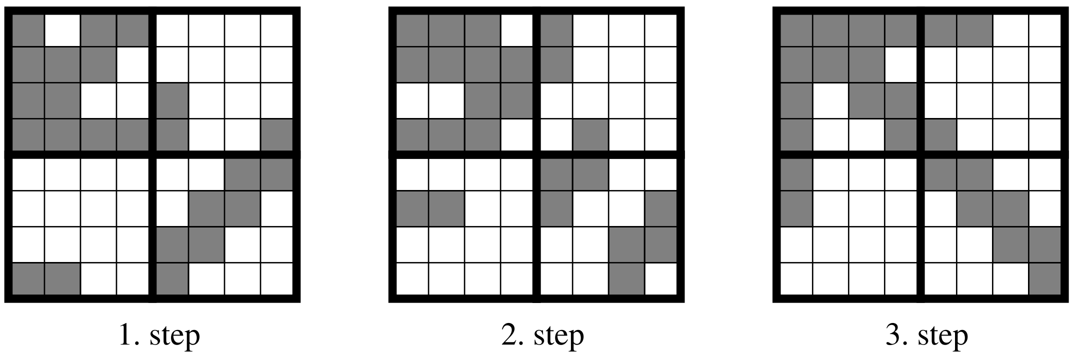
```
BEGIN
  WHILE pic_size > 1 DO
    size2 := pic_size div 2;
    PARALLEL
      y := dim1 mod pic_size;
      x := dim2 mod pic_size;
      IF x < size2 THEN propagate.up  ^size2 (color,buffer);
                        IF y >= size2 THEN b2 := buffer END
                   ELSE propagate.down^size2 (color, buffer);
                        IF y <  size2 THEN b2 := buffer END
      END;
      IF y < size2 THEN propagate.left^size2 (color, buffer);
                        IF x <  size2 THEN b2 := buffer END
                   ELSE propagate.right^size2 (color, buffer);
                        IF x >= size2 THEN b2 := buffer END
      END;
      color := b2;  (* copy new value *)
    ENDPARALLEL;
    pic_size := size2
  END (* while *)
END rotate;


BEGIN
  ...                   (* load picture    *)
  rotate(m_size);       (* rotate picture  *)
  ...                   (* display picture *)
END image_rot.
```

Stepwise Algorithm Illustration:


Initial Picture


Cyclic data transfer during the first step

In the initial picture, the "arrow" points to the lower left. Divided into four quadrants, the image data is cyclically being exchanged. The next recursive steps will be in parallel with decreasing side length.



While the first step produces an apparent mess, the following steps give the desired result. The "arrow" now points to the upper left corner. For n pixels, we needed $\log_2(n)/2$ steps to complete the algorithm.

Figure 10.4: Illustrating the Rotate-Algorithm

10.3 Prime Numbers

The sieve of Eratosthenes is one of the classical demonstrations of parallel algorithms. Please note that there are no connections needed to solve this problem!

Program 4: Generating prime numbers with the sieve of Eratosthenes

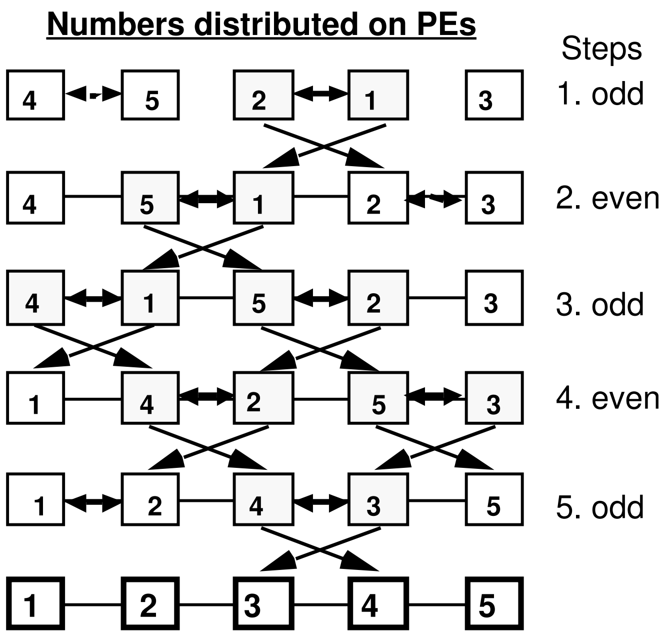
```
SYSTEM Sieve;
CONFIGURATION List [1..1000];
CONNECTION  (* none *);

SCALAR  prime    : integer;
VECTOR  candidate: boolean;

BEGIN
   PARALLEL
      candidate := id_no >= 2;
      WHILE candidate DO
         prime := REDUCE.First(id_no);
         WriteInt(prime, 5);
         IF id_no MOD prime = 0    (* remove multiples *)
            THEN candidate := FALSE
         END
      END
   ENDPARALLEL
END Sieve.
```

The `while` loop will be executed as long as at least a single candidate-PE remains, i.e. the vector `candidate` has value `true` for at least one PE. All PEs excluded from the `while` loop are disabled. The `if`-selection removes all multiples of a prime in **constant** time.

10.4 Sorting

Odd-Even Transposition Sorting (OETS), a.k.a. *"parallel bubble-sort"* is a well-known parallel algorithm in the literature. With n processors it is possible to sort n numbers in time O(n). All PEs are linked in a linear chain. The algorithm executes n steps, each consuming constant time. During odd steps, the PE pairs 1-2, 3-4, and so on are compared in parallel, while in even steps the PE pairs 2-3, 4-5, and so on are handled.

Each PE holds one component of the vector `val` to be sorted as well as copies of the left and right neighbor's values. The vector `swap` is a marker for exchanges occurred which have to be completed at the right neighbor-PE.



Program 5: Odd-Even Transposition Sorting

```
SYSTEM Sort;
CONST  n = 1000;
CONFIGURATION list [1..n];
CONNECTION  left : list[i]  ->  list[i-1].right;
            right: list[i]  ->  list[i+1].left;
SCALAR  k       : integer;
VECTOR  val,r,l : integer;
        swap    : boolean;

BEGIN
... (* read input data *)
  FOR k:=1 TO n DO
    PARALLEL
      PROPAGATE.right(val,l);
      PROPAGATE.left (val,r);
      (* l/r hold the value of the left/right neighbor *)
      swap := false;
      IF odd(k) THEN  (* compare 1-2, 3-4, ... *)
```

```
          IF odd(dim1) AND (r < val) THEN
            val  := r;
            swap := true
          END
        ELSE  (* even (k)  compare 2-3, 4-5, ... *)
          IF even(dim1) AND (r < val) THEN
            val  := r;
            swap := true
          END;
        END;

        PROPAGATE.right(swap);
        IF swap AND (id_no > 1) THEN val := 1 END;
      ENDPARALLEL
    END;
  ... (* write output data *)
  END Sort.
```

A more sophisticated version of the same program is shown in program 6. In this version by Claus Brenner from Universität Stuttgart, the connection structure is adapted ideally for the odd/even data exchange. Therefore, it requires only one third of the propagate operations of program 5.

Program 6: An Optimized Version of Odd-Even Transposition Sorting

```
SYSTEM OETSort;
CONST           n = 10;
CONFIGURATION   chain [1..n];
CONNECTION
         oddout:  chain[i] -> {odd(i) } chain[i+1].oddin,
                              {even(i)} chain[i-1].oddin;
        evenout: chain[i] -> {even(i)} chain[i+1].evenin,
                              {odd(i) } chain[i-1].evenin;
SCALAR  k         : INTEGER;
VECTOR  val, temp: INTEGER;

BEGIN
  ... (* read input data & LOAD *)
  FOR k:=1 TO (n+1) DIV 2 DO
    PARALLEL
     (* ODD step  *)
     propagate.oddout(val,temp);
     IF odd(id_no)  = (val > temp) THEN val:=temp END;

     (* EVEN step *)
     propagate.evenout(val,temp);
     IF even(id_no) = (val > temp) THEN val:=temp END;
   ENDPARALLEL
  END;
  ... (* STORE & write output data *)
END OETSort.
```

10.5 Numeric Integration

The value of   may be approximated by numeric integration as shown in the formula and the diagram of fig. 10.5. The Parallaxis program is simple and explains itself; there are no connections required between PEs. This algorithm was used as a reference by R. Babb [Babb 88]. However, since the problem is fairly simple and does not need any inter-processor communication at all, its value as a reference is most doubtful!

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

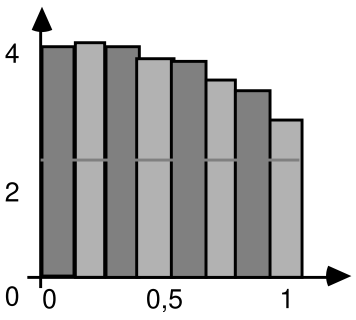


Figure 10.5: Approximation of pi

Program 7: Approximating pi by numeric integration

```
SYSTEM compute_pi;
CONST interval = 1000;
              width    = 1.0 / float(interval);
CONFIGURATION list [1..interval];
CONNECTION (* none *);

VECTOR val: real;

PROCEDURE f (VECTOR x: real): VECTOR real;
(* function to be integrated *)
BEGIN
  RETURN(4.0 / (1.0 + x*x))
END f;

BEGIN
  PARALLEL
    (* integral approximation with rectangle-rule *)
    val := width * f( (float(id_no)-0.5) * width );
  ENDPARALLEL;
  WriteReal(REDUCE.sum(val), 15);
END compute_pi.
```

10.6 Cellular Automata

A cellular automaton almost perfectly reflects Parallaxis' model of an SIMD hardware. We have a number of independently operating entities, all with the same "program" and interconnected by a characteristic symmetrical pattern of links. The program below produces one of these nice triangular patterns. Other automata, like the "Game of Life" or the "Hodge-Podge-Machine" may be implemented with simple modifications to this program, reflecting the new building rules.

Program 8: A simple cellular automaton

```
SYSTEM cell;
CONST n = 79;   (* number of elements *)
      m = 50;   (* number of steps    *)
CONFIGURATION list [1..n];
CONNECTION left:  list[i] -> list[i-1] .right;
           right: list[i] -> list[i+1] .left;

SCALAR i       : integer;
VECTOR val,l,r: boolean;
...
BEGIN
  PARALLEL (* Init *)
    val := false;
  ENDPARALLEL;
  PARALLEL [n div 2] (* middle *)
    val := true;
  ENDPARALLEL;

  FOR i:= 1 TO m DO
    out; (* omitted / to display current state *)
    PARALLEL
      propagate.left (val,l);
      propagate.right(val,r);
      val := l<>r;
    ENDPARALLEL;
  END;
END cell.
```

10.7 Fractal Geometry

The following parallel algorithm is based on the sequential algorithm for constructing a fractal curve by midpoint displacement which is shown by Peitgen and Saupe in "The Science of Fractal Images". The parallel version developed by Thomas Bräunl uses a tree structure to generate the fractal curve at increasing detail. Again, the recursive nature of the problem could be expressed by a simple parallel iteration in procedure "MidPointRec".

Program 9: Generating a fractal curve

```
SYSTEM fractal;
CONST  maxlevel = 7;
(* binary tree structure *)
CONFIGURATION tree [1..2**maxlevel -1];
CONNECTION    son_l : tree[i] -> tree[2*i].father;
              son_r : tree[i] -> tree[2*i+1].father;
             father: tree[i] -> {even(i)} tree[i div 2].son_l,
                                {odd(i)}  tree[i div 2].son_r;
```

```
SCALAR  i,j          : integer;
        delta        : real;
        field        : ARRAY [1..2**maxlevel-1] OF real;
VECTOR  x, low, high: real;

PROCEDURE Gauss(): VECTOR real;
(* special random number generator, not further discussed
here *)

PROCEDURE MidPointRec(SCALAR delta: real; SCALAR level:
integer);
SCALAR  min, max, max2 : integer;
BEGIN
  (* select tree level: 2^(level-1) <= id_no <= 2^level -
1 *)
  min := 2**(level-1);
  max := 2 * min - 1;
  max2:= 2 * max + 1;

  PARALLEL
    (* current tree level only *)
    IF min <= id_no <= max THEN
      x := 0.5 * (low + high) + delta*Gauss();
    END;

    (* select next level also for data propagation *)
    IF min <= id_no <= max2 THEN
      propagate.son_l(low);
      propagate.son_r(high);
      propagate.son_l(x);
      propagate.son_r(x);
      IF even(id_no) THEN high:=x ELSE low:=x END;
    END
  ENDPARALLEL
END MidPointRec;

BEGIN (* main program *)
  Init;
  FOR i:=1 TO maxlevel DO
    delta := 0.5 ** (float(i+1)/2.0);
    MidPointRec(delta,i);
  END;
  ...  (* output data *)
END fractal.
```

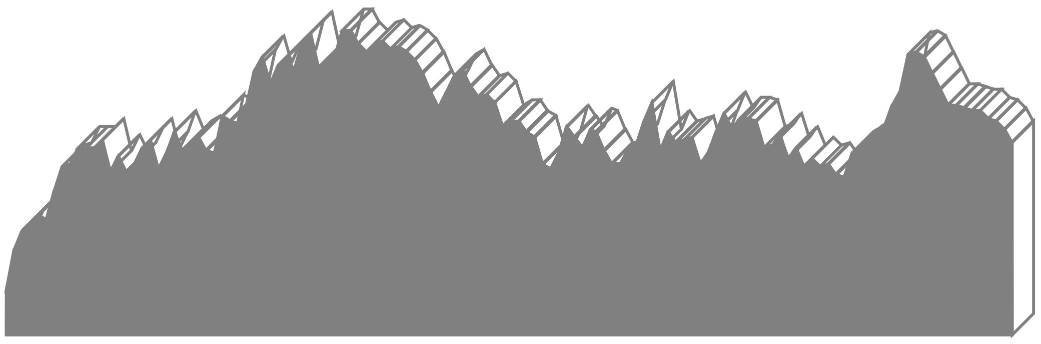


Figure 10.6: Sample fractal curve

The curve displayed in fig. 10.6 was created by using the program above. The program uses a Gaussian-weighted random function not further discussed here.

10.8 Systolic Matrix-Multiplication

Fig. 10.7 explains the systolic behaviour of the matrix multiplication algorithm implemented below. The algorithm idea was taken from Shapiro's "Concurrent Prolog" and translated into Parallaxis. But only after modifications by Ingo Barth, the program produced reasonable performance results.

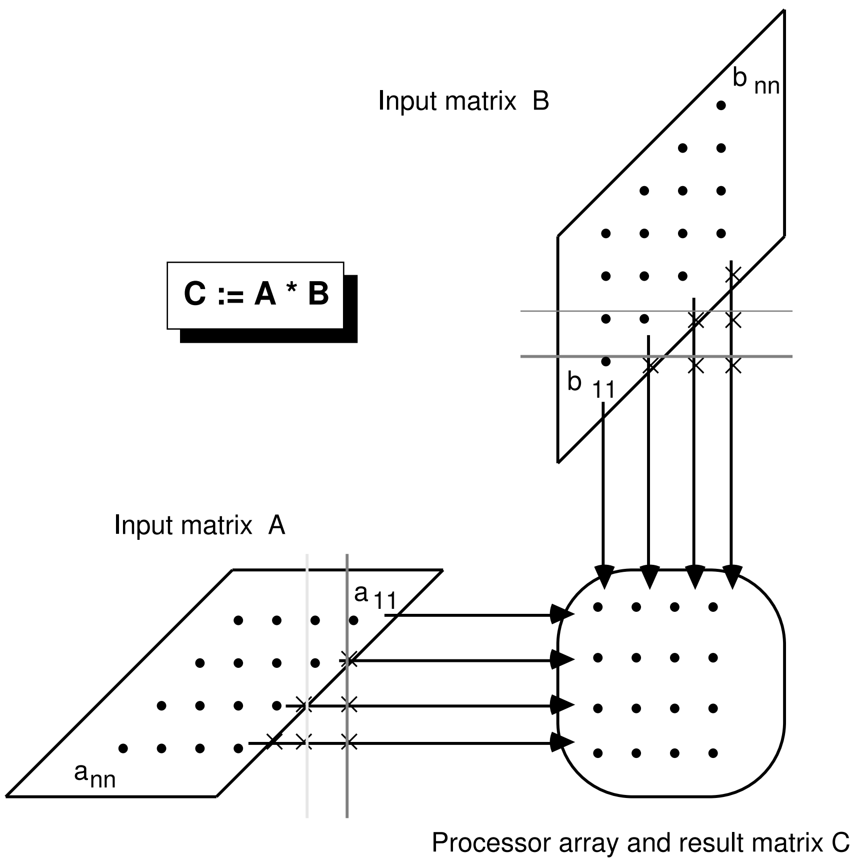


Figure 10.7: Computation Schema of systolic matrix multiplication

Program 10: Systolic Matrix Multiplication

```
SYSTEM  systolic_array;
CONST max    = 10;
TYPE  matrix = ARRAY [1..max],[1..max] OF REAL;
CONFIGURATION  grid [max],[max];
CONNECTION
   left   : grid[i,j]  ->  grid[i,(j-1) MOD max].left;
   up     : grid[i,j]  ->  grid[(i-1) MOD max,j].up;
   shiftA : grid[i,j]  ->  grid[i,(j-i) MOD max].shiftA;
   shiftB : grid[i,j]  ->  grid[(i-j) MOD max,j].shiftB;

SCALAR i,j         : INTEGER;
       a,b,c       : matrix;

PROCEDURE matrix_mult(SCALAR VAR a,b,c : matrix);
(* c := a * b *)
SCALAR k: INTEGER;
VECTOR ra,rb,rc : REAL;
BEGIN
  LOAD (ra,a);
  LOAD (rb,b);
  PARALLEL
    PROPAGATE.shiftA(ra);
    PROPAGATE.shiftB(rb);
    rc := ra * rb;
    FOR k := 2 TO max DO
      PROPAGATE.left(ra);
      PROPAGATE.up(rb);
      rc := rc + ra * rb;
    END;
  ENDPARALLEL;
  STORE(rc,c);
END matrix_mult;

BEGIN
  ...  (* input matrices *)
  matrix_mult(a,b,c);
  ...  (* output result  *)
END systolic_array.
```

11 Parallaxis Applications

Since the first release of Parallaxis, we had quite a large number of students who developed Parallaxis applications as their term project or Master's thesis. Here, we would like to introduce a few of this growing number. We do not repeat the source code, as we did in chapter 10, but we will give some information on each individual application. All application programs are available from our ftp-server or on floppy disk.

11.1 Stereo Vision

The processing of images is an application which is ideally suited for data level parallelism. An image is stored as an array of picture elements, or pixels. E.g., a picture with 256 pixels width and 256 pixels heigth has a total of 65,536 data elements. Each of them contains information about color and brightness. Processing an image normally means to perform a quite simple operation on each pixel or a very small region around it. This can be done very efficiently on massively parallel computers like the Connection Machine or the MasPar in combination with a parallel programming language like Parallaxis.

As a first step to compute depth information, we decided to work with random-dot stereograms. In these pictures, the monocular information is completely absent and they have perfectly controlled properties to find out how binocular localization resolves ambiguities. However, right now we are also working with scanned real world pictures, especially aerial pictures.

The figures below show such a pair of stereograms together with their solution computed by a Parallaxis program. The two random dot arrays are identical except for a pattern which is horizontally shifted several pixels in one array. People who are able to "cross their eyes" (by focussing to a distant point) can recognize the pattern without tools (prism or red-green glasses).

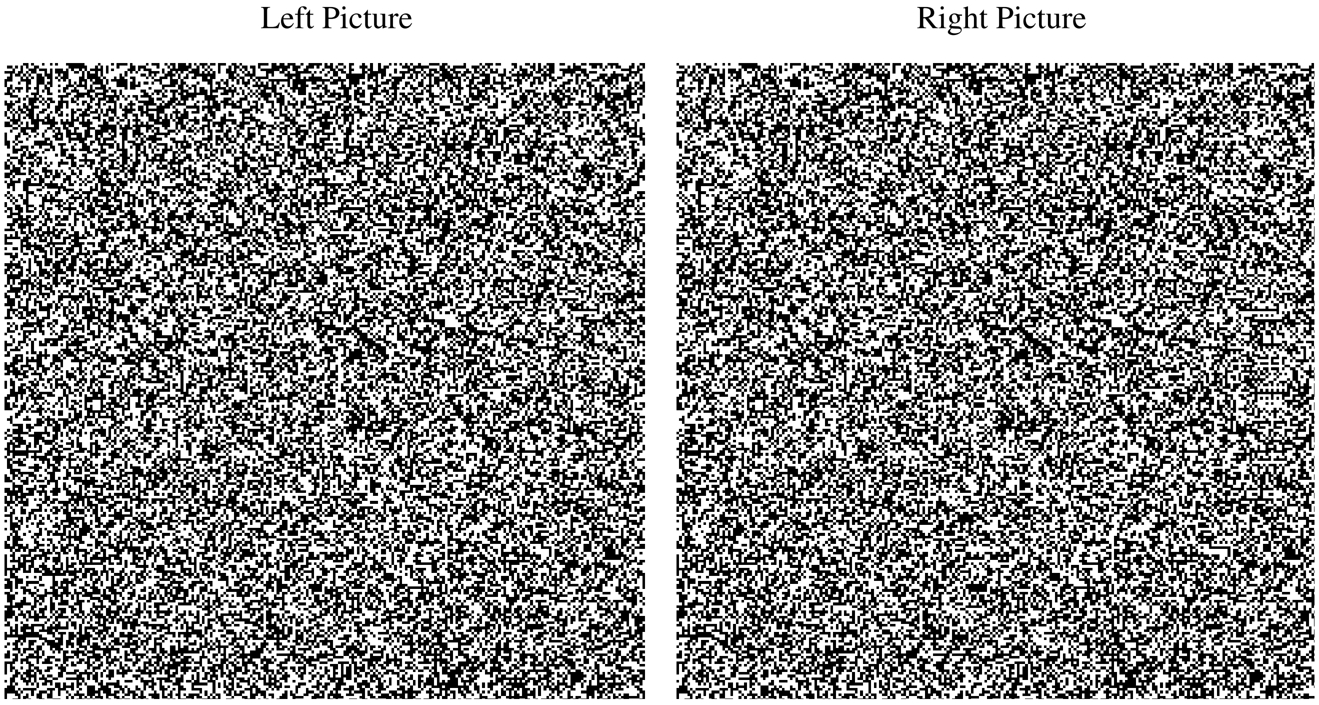


Figure 11.1: Stereo Image Pair

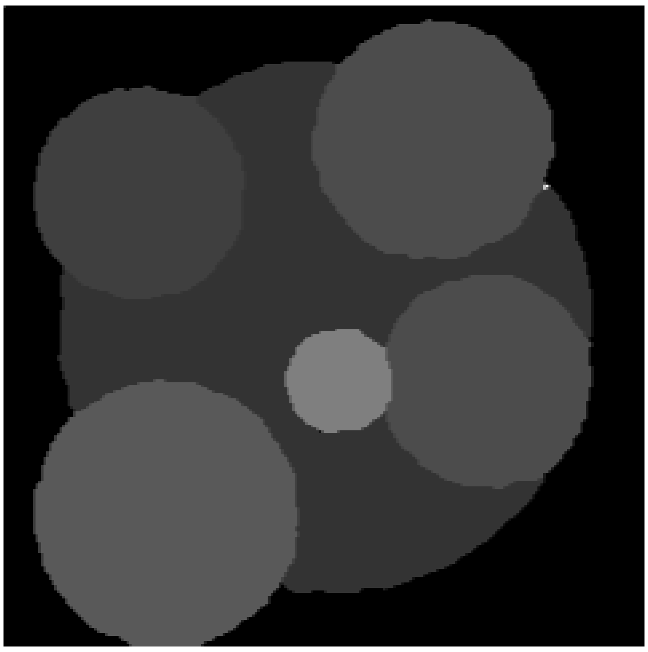


Figure 11.2: Computed Depth Information

The depth perception algorithm works as follows. First, both arrays are mapped one over the other. Then, the right eye's array is shifted stepwise to the right. At each step, every pixel calculates whether the current level is a possible depth position. After the shifting phase, each pixel may have several possible depth locations. Checking also the neighbor-pixels' results allows a rather exact determination of which depth level it belongs to.

Now, let us take a look at the efficiency of this algorithm. The simulation of the example shown requires 57,600 virtual processing elements. Only the major routines were measured, input and output was disregarded. The number of program steps is quite low, while the **overall processor load of 74%** gives extremely good results! Assuming your parallel computer performs one assembly command in a microsecond, it would take less than 1.5 milliseconds to compute the complete stereo image.

The stereo matching algorithm has been programmed by Karsten Krauskopf [Krauskopf 90].

11.2 Hidden-Surface and Ray Tracing Algorithms

Another field of applications surveyed in Parallaxis are graphics algorithms. Among this group of algorithms especially the hidden surface algorithms have been examined. These have the purpose to determine the visible surfaces from a given three-dimensional scene and project them on an image plane. There are algorithms which examine the surfaces in three-dimensional space, such as the Z-buffer algorithm, and other algorithms, e.g. scan line, which determine the visibility in image space.

These algorithms, as described in Foley, van Dam: "Introduction to Computer Graphics", are being examined with regard to parallelism. The Z-buffer algorithm works with a two-dimensional field of memory, which has one entry for each pixel, containing the actual color and the Z-value. For all surfaces successively, each surface pixel's depth value is compared to the local

Z-buffer value and, if appropriate, color and Z-value are adjusted. This problem is solved in parallel with a field of processor elements (PE), each representing a single pixel. So all pixels covered by one polygon can be examined in a single step and the task is completed in time O (p) for p polygons.

Another possibility is to employ one PE per polygon, to determine first the covering polygons and then the visible polygon for each pixel in parallel. This step takes time O ($\log_2(p)$) for each pixel so the whole task needs time O ($n^2*\log_2(p)$).

With regard to the time complexity, it is better to take the first approach, because the run time is only proportional to the number of polygons. But in parallel processing also the number of active PEs in each step is an important aspect. So in scenes with a very homogeneous distribution of small polygons, the second algorithm is better suited, because all PEs have to determine for each pixel whether their polygons are active.

Scan line algorithms work on all polygons at once, processing one image line after another. Each scanline is divided by crossing edges (so called active edges) into spans. When processing a line, the color has to be determined only at the beginning of a span, because the whole span has the same color. The information of active polygons inside each span has to be moved from one span to the next one, so that for each span the active polygons are known. Because of this dependency, processing of spans in parallel does not give good load results. However, it is obvious that we can work on all scan lines in parallel, using one PE for each line. So the overall time needed to solve this problem is proportional to the time needed to process the most complex scan line.

Ray tracing algorithms can be parallelized as well. In our approach, each PE traces a single ray into the scene. We can handle spheres and polygons in the scene file represented in NFF (Haines' Neutral File Format). The overall time for ray tracing a scene is the time needed for the most complex ray to compute.

Two sample images created by a Parallaxis hidden-surface program and ray tracing program are shown below.

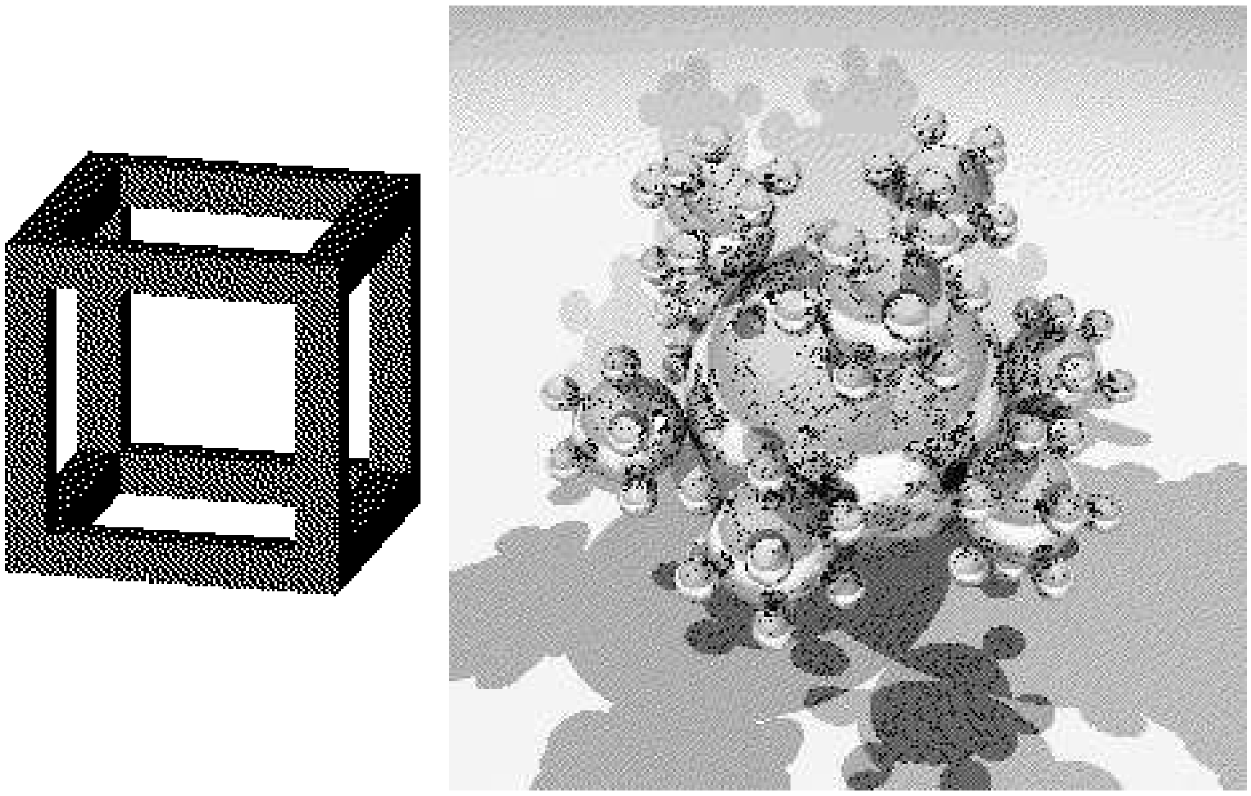


Figure 11.3: rendered images from scene descriptions

The hidden-surface and ray tracing algorithms have been programmed by Sabine Liebelt [Liebelt 91].

11.3 Solving Linear Equation Systems

This application program solves a linear equation system by applying a massively parallel version of the Gaussian elimination algorithm. There are n × (n+1) PEs used, one for each coefficient. The program operates in two phases: first the transformation of the coefficient matrix into a triangular matrix, then the resubstitution of the component solutions from back to front.

Two kinds of parallel algorithms have been implemented and tested: the Gaussian elimination and the Gauss–Jordan method. Two different kinds of algorithms for the Gaussian elimination are implemented, one with few processor elements O(n) and one with many processor elements O(n^2). For the Gauss Jordan method two algorithms have been implemented, also with differences in the number of processor elements. The Gaussian elimination and the Gauss Jordan method are similar, the differences of the algorithms are shown in the table below. Only the Gauss algorithm is included in the Parallaxis applications.

The test results are as follows:

Gauss Algorithm	**version 1**	**version 2**
number of PEs	650	26
time steps	44,928	264,324
program steps	21,355	118,299
average of use the PE in percent	37.8	49.6

The linear equation solver has been programmed by Reinhard Verba [Verba 90].

11.4 Fast Fourier Transformation

The Fast Fourier Transformation requires complex valued multiplications and additions. The algorithm distributes the n input values to the n PEs, computes the solutions locally for each PE with an iteration, and returns these solutions. The time required for these computations is O(n*-log(n)).

The discrete Fourier transform and the fast Fourier transform have been compared in the table below.

The test results are as follows:

Fourier Algorithms	**FFT**	**DFT**
number of PEs	64	4096
time steps	26,621	16,249
program steps	14,249	4,525
average of use the PE in percent	55.2	22.1

This shows that the DFT is faster because of the use of the larger number of processor elements, whereas the fast Fourier transform is more efficient in average PE usage (load). Only the FFT algorithm is included in the Parallaxis applications.

The fast fourier transformation algorithm has been programmed by Reinhard Verba [Verba 90].

11.5 Simulated Annealing

Simulated Annealing is an heuristical approach for solving NP-complete problems. The best parameter values of a complex system are searched in a local neighborhood area. To prevent being caught in a local minumum, a temperature function is added. In the beginning, the system is heated up, allowing a high percentage of random moves that do not have to improve the parameters. While gradually decreasing the temperature towards the freezing point, less and less random moves are allowed, and the system is slowly converging to lower energy values, representing better results.

Since the structure of this heuristics is sequential (we cannot compute the cooling phase before heating up the system), we applied the same structure to our SIMD Parallaxis program. The only chance to apply massive parallelism, is in the computation of the energy function. Several problems suited for simulated annealing have been studied with this parallel program. The results were that using a massively parallel architecture for simulated annealing pays off only for very complex problems, like the chip placement problem with a high degree of interdependencies. The less complex travelling salesperson problem may as well be computed on a sequential hardware with simulated annealing. The simplicity of the energy function (here: the tour length) does not require a large number of PEs

The simulated annealing algorithm has been programmed by Volker Walter [Walter 91].

11.6 The n-Body Problem

Lastly, the n-body problem is to simulate a number of n masses in space, influencing each other (e.g. this could be the sun and the planets of our solar system). All n bodies are linked with the appropriate n^2 interactions. The Parallaxis implementation of this problem gives a good illustration for using multiple configurations (here: bodies and interactions) and multiple connections (1:n) in a program.

The n-body problem has been derived from an algorithm Rose and Steele in [Rose Steele 87] by Frank Sembach with some amendings by Ingo Barth.