Published in: Australian Journal of Intelligent Information Processing Systems (AJIIPS), vol. 6, no. 3, 2001, pp. 164–174 (11)

Tutorial in Data Parallel Image Processing

Thomas Bräunl
Dept. of Electrical and Electronic Engineering
Centre of Intelligent Information Processing Systems
The University of Western Australia, Nedlands, Perth WA6907
email: braunl@ee.uwa.edu.au

Abstract

An overview over data parallel image processing routines is given. The focus of this tutorial is on real time, low level image processing for parallel active vision systems. Image operator classes discussed are point operators, local operators, dithering, smoothing, edge detection, morphological operators, and image segmentation.

1. Introduction

Image processing is gaining larger importance in a variety of application areas. Active vision, e.g. for autonomous vehicles, requires substantial computational power, in order to be able to operate in real time. Here, vision allows the development of more flexible and intelligent systems than any other sensor system. In addition, there is also the need to speed up non-critical image processing routines, e.g. in evaluating medical or satellite image data.

Basic image processing routines are very well suited for synchronous parallel processing. While the era of large-scale SIMD systems like Connection Machine [11] or MasPar [9] may have passed, the concept may well be used for small embedded systems. The ideal concept of having one processor (ALU) per image pixel allows a very simple and natural definition of image operations.

In this article, we would like to give an extensive overview of typical basic image processing operations, demonstrating how they can be programmed in data parallel mode.

The notation we use for synchronous parallel programming is the author's Parallaxis-III programming language [3]. While the language's syntax and semantics is defined elsewhere [4] and shall not be repeated here, a small number of basic features present in virtually all SIMD programming languages shall be discussed shortly.

Active Set. All languages provide control structures to select the set of active PEs for an operation. This is required, since in SIMD mode only one operation can be executed by all PEs at a time – unless they are inactive. Especially, PEs being exempted from an operation cannot be used for performing another operation at the same time. In Parallaxis, selecting the active set is implicit for each selection- or loop-statement involving vector data in its boolean condition. These conditions may also include positional data, so PEs can easily be selected, e.g. by their row and column number.

Processor Groups. Parallaxis, unlike some other SIMD languages, allows the declaration of PE configurations (groups of processors), together with their arrangement in an n-dimensional cube. Static connections for data exchange between PEs may also be specified at compile time, while dynamic connections may be added data-dependent at run time. In the context of image processing, it is sufficient to know that there are connections defined as *up*, *down*, *left*, and *right*, to facilitate data exchange between PEs.

Data Exchange. Parallaxis provides three operations for data exchange between PEs. In its simplest version, an expression is created by moving data into the specified direction of a previously defined connection. Parallaxis prevents the appearance of undefined data at border PEs, so for image processing, no special border treatment is required.

Vector Reduction. Besides sequential data transfer between host and vector PEs, Parallaxis provides a reduction function, in order to reduce a vector value to scalar value in logarithmic time – depending on the system architecture. Operators used in reduction are either predefined or implemented by the application programmer.

When we started working on parallel image processing, we first took a look at conventional sequential low level image processing routines, as defined in several textbooks [6],[7],[10]. There is quite a large number of basic routines, which can be used as building blocks for larger applications.

- Point Operators
- Local Operators
- Global Operators
- Histograms
- Edge Detection
- Edge Thinning
- Corner Detection
- Regions
- Hough Transform
- Fast Fourier Transform
- Stereo Vision
- Motion Detection
- Textures with Cooccurance

It turned out that each of these applications is very well suited for an efficient synchronous parallel implementation. This led to the development of two textbooks [2], [4].

2. Parallel Image Representation

We assume a two-dimensional array of PEs, large enough to provides one PE per image pixel. In case of fewer PEs, we can still have one *"virtual PE"* per pixel, with the operating system or the compiler taking care of the iteration required.

This architecture is reflected in our definition in Program 1. The actual PE field (and image) size is left open to be specified by the application programmer. The symbolic names `right`, `left`, `up`, `down`, and the four diagonals may conveniently be used in subsequent data exchange

Program 1: Processor configuration for image data

```
1 CONFIGURATION grid[*],[*];
2 CONNECTION
3 right:grid[i,j]<->grid[i  ,j+1]:left;
4 up   :grid[i,j]<->grid[i-1,j  ]:down;
5 up_l :grid[i,j]<->grid[i-1,j-1]:down_r;
6 up_r :grid[i,j]<->grid[i-1,j+1]:down_l;
```

statements. As shown in Figure 1, the *y* coordinate grows from top to bottom, in order to comply with standard image file formats and screen graphics routines like X windows.

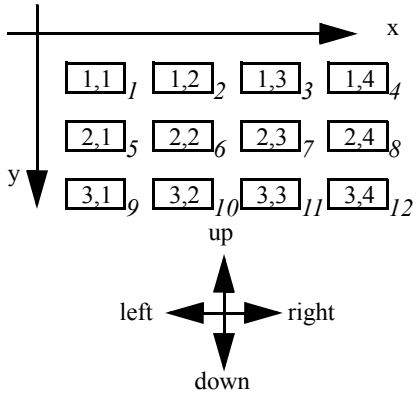

Figure 1: Processor arrangement

We will now define the parallel image data types. Like in sequential image processing, we distinguish between color images, grayscale images, and binary images. Constants for black and white have to be specified for each type. Details can be seen in Program 2.

Program 2: Data types

```
 1 TYPE binary = BOOLEAN;
 2      gray   = [0..255];
 3      color  = RECORD
 4                red, green, blue: gray
 5               END;
 6
 7 CONST b_black =  TRUE;
 8       b_white = FALSE;
 9       g_black =     0;
10       g_white =   255;
11       c_black = color(  0,  0,  0);
12       c_white = color(255,255,255);
```

3. Point Operators and Local Operators

In the following, we will present a number of basic image processing operators together with their data parallel implementation. As we believe, the parallel notation will in most cases be even simpler and more readable than its sequential counterpart. The typical header of a point operation looks like:

```
PROCEDURE xyz(img: VECTOR OF gray):
                  VECTOR OF gray;
```

Whereas the typical header of a local operation looks like:

```
PROCEDURE xyz(img: grid OF gray):
                  grid OF gray;
```

In both cases, one grayscale image is converted into another. Using the keyword `VECTOR` defines a parameter of any vector data type, so this declaration is even more general than the previously defined configuration `grid`. As a logical consequence, local data exchange inside the procedure is only possible if parameters use configuration `grid`, whereas `VECTOR` hides the connection structure.

3.1 Point Operators

The simplest class of image operators are point operators. A new pixel is computed as a function of the original pixel; no data of neighbor pixels is used. Therefore, point operators do not have any data dependence and can be easily expressed in parallel.
In some application areas, like medical imaging or the processing of satellite data, images may stretch only over a limited band of the grayscale. Therefore, these images have low contrast and details are difficult to recognize [1]. In order to improve contrast, the image's grayscale band can be stretched to full range (here 0..255). Figure 2 sketches the procedure.

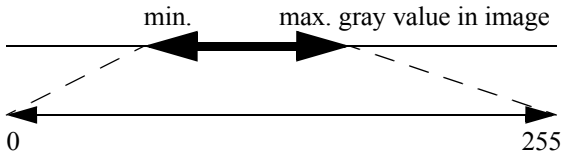

Figure 2: grayscale stretch

Program 3 implements the general grayscale stretching, using parameters `g_min` and `g_max` instead of constants 0 and 255. First, the max and min grayscale values of the whole image are determined by using the `REDUCE` operation. Alternatively, these values could also have been supplied as parameters to the procedure. Care has been taken to avoid division by zero in case of a monochromatic image. Then, in the `RETURN` statement, the image is shifted in grayscale range by subtraction `(img-tmin)` to range `0..(tmax-tmin)`. The division by `(tmax-tmin)` shifts it to range `0..1`, wile the multiplication with `(g_max-g_min)` brings it to range `0..(g_max-g_min)`. The final addition of `g_min` stretches the image to the desired range `g_min .. g_max`.

3.2 Average and Median

All following operators are local operators, that is in order to compute one pixel, a number of neighbor pixels also have to be taken into account. Several neighborhood areas are possible, e.g. only left/right and up/down neighbors (involving 5 pixel), a full 3×3 neighborhood (9), 5×5 (25) or up to 11×11 (121). Of course, the larger the neighborhood considered, the more time-consuming will the execution of these operators be. Access to neighbor pix-

Program 3: grayscale stretch

```
 1 PROCEDURE gray_stretch(img:
 2   VECTOR OF gray; g_min,g_max: gray):
 3           VECTOR OF gray;
 4 (* stretch values to g_min..g_max *)
 5 VAR tmax,tmin: INTEGER;
 6 BEGIN
 7   tmin := REDUCE.MIN(img);
 8   tmax := REDUCE.MAX(img);
 9   (* avoid division by 0 *)
10   IF tmin = tmax THEN INC(tmax) END;
11   RETURN (g_max-g_min) * (img-tmin)
12          DIV (tmax-tmin) + g_min;
13 END gray_stretch;
```

els is implemented by local data exchange in the data parallel model.

A simple application of local operators are image smoothing and noise reduction. The *average* operator simply computes the average grayscale value of a pixel and its neighbors. The basic operation for averaging is adding the grayscale value of a pixel and all of its neighbors. Figure 3 shows this for a 3×3 neighborhood.

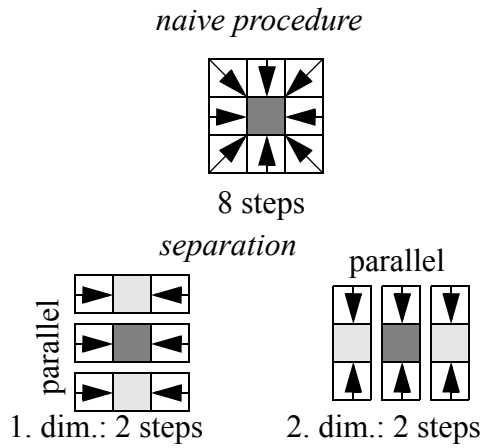

Figure 3: Separation of local operators

While the naive procedure requires eight steps to exchange data with all neighbors, half of this time can be saved by applying separation between *x*- and *y*-dimension. First, the neighbor pixels within a row are added, then the *partial result* is moved between rows and is added to form the total sum. Separation can be applied for all symmetric local operators. In general, a local $n \times n$ operator requires n^2-1 steps, while separation reduces the cost to $2*(n-1)$. The implementation of grayscale averaging in Program 4 now becomes straightforward.

The *median* operator averages in a different way. Instead of computing the local average, it selects the middle element of a local neighborhood, sorted by their gray values. Instead of presenting the true *median* filter, we apply a small simplification, which allows us to do a parallel separation. We call this adapted filter *"Fast-Median"*. While the true *median* computes the middle value of all neighbors, our *fast-median* determines the middle (over all rows) of the middle (over all columns) gray value.

Program 4: Local sum and average

```
 1 PROCEDURE sum_3x3(img:
 2    grid OF gray): grid OF INTEGER;
 3 (* returns sum of local 3x3 area *)
 4 VAR res: grid OF INTEGER;
 5 BEGIN
 6   res:= img + MOVE.right(img) (*hor.*)
 7             + MOVE.left(img);
 8   res:= res + MOVE.down(res)  (*ver.*)
 9             + MOVE.up(res);
10   RETURN res;
11 END sum_3x3;
```

```
 1 PROCEDURE average_3x3(img:
 2    grid OF gray): grid OF gray;
 3 (* average value of 3x3 area *)
 4 BEGIN
 5   RETURN sum_3x3(img) DIV 9
 6 END average_3x3;
```

Example

1	7	4
3	1	99
6	0	2

Average: (1+7+4+3+1+99+6+0+2) / 9 ≈ **14**

Median: middle (0, 1, 1, 2, 3, 4, 6, 7, 99) = **3**

Fast-Median: middle**(**middle(1,7,4),middle(3,1,99),middle(6,0,2)**)**
= middle(4,3,2) = **3**

Figure 4: Average and median

Figure 4 explains the differences between *average*, *median*, and *fast-median*. It should be noticed that *average* is much more sensitive to extreme-valued pixels than *median*. When applied to real word images, there is almost no perceivable difference between *median* and *fast-median*.

Program 5 shows the implementation of *fast-median*. Like for the previous operators, separation is used to save execution time. First, the middle element within the same line is used, then this partial result is moved across the lines to find the middle element of the middle elements. Sorting of three numbers is achieved simply by applying compare-and-swap three times in a row.

Figure 5 shows the results of applying *average* and *fast-median* operator to a noisy image. The noise has been artificially added by inserting black and white pixels at random positions *("salt and pepper noise")*.

While the *average* operator merely smears the error pixels over a larger area, the *fast-median* operator is actually able to reduce the noise while preserving most of the image resolution.

3.3 Dithering

Dithering is used to transform a grayscale image into a binary image, while preserving some grayscale information at the cost of losing some image resolution. Each grayscale value of the original image is translated into a pattern of binary values in the output image.

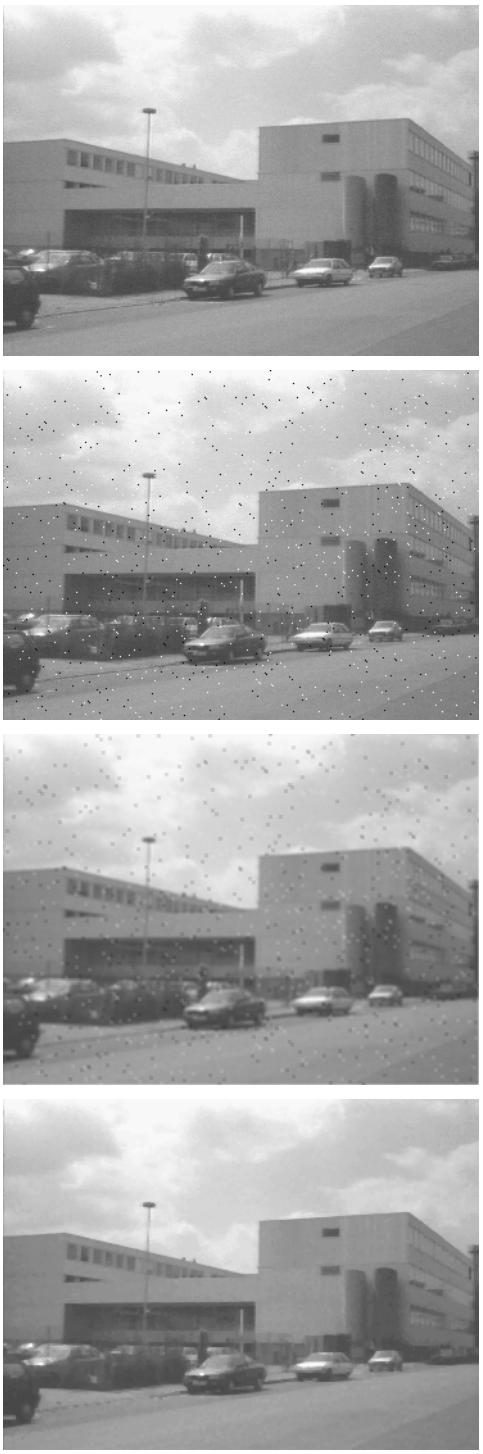

Figure 5: Original image, image with noise added, average operator, fast-median operator

The simplest approach is *ordered dithering* [5]. Our parallel implementation uses 2×2 patterns, thus being able to distinguish five different gray values as shown in Figure 6.

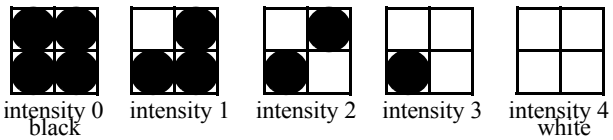

Figure 6: Dithering with 2×2 pattern

Program 5: Fast-Median

```
 1 PROCEDURE median_3x3fast(
 2   img: grid OF gray): grid OF gray;
 3 (* approximation: median in y-dir.
 4    of median in x-direction *)
 5 VAR a: grid OF ARRAY[1..3] OF gray;
 6 BEGIN
 7  a[1] := MOVE.left (img);
 8  a[2] := img;
 9  a[3] := MOVE.right(img);
10  (* sort 3 elems. with 3 comp/swaps *)
11  IF a[1] > a[2] THEN
12    swap(a[1],a[2]) END;
13  IF a[2] > a[3] THEN
14    swap(a[2],a[3]) END;
15  IF a[1] > a[2] THEN
16    swap(a[1],a[2]) END;
17  (* send median in x-dir. up/down *)
18  SEND.up  (a[2],a[1]);
19  SEND.down(a[2],a[3]);
20  (* sort 3 elems. with 3 comp/swaps *)
21  IF a[1] > a[2] THEN
22    swap(a[1],a[2]) END;
23  IF a[2] > a[3] THEN
24    swap(a[2],a[3]) END;
25  IF a[1] > a[2] THEN
26    swap(a[1],a[2]) END;
27  RETURN a[2];
28 END median_3x3fast;
```

Program 6 implements *ordered dithering* in data parallel. Please note, that only one quarter of all pixels is considered for this transformation, as activated by the initial `IF`-selection. Therefore, it may be useful to perform an *average* operator before applying *ordered dithering*. According to the patterns in Figure 7, the procedure determines by a pixel's gray value which of the four fields should be black (`TRUE`) and which should be white (`FALSE`). Data exchange is done with the `SEND` procedure, which differs from `MOVE` in the way that it does not require the receiver to be active.

Program 6: Ordered Dithering

```
 1 PROCEDURE dither_ordered(
 2   img: grid OF gray): grid OF binary;
 3 (* ordered dithering, 2x2 patterns *)
 4 CONST thres = g_white DIV 5;
 5 VAR res: grid OF binary;
 6 BEGIN
 7   IF ODD(DIM(grid,2)) AND
 8      ODD(DIM(grid,1)) THEN
 9     res := img < thres;
10     SEND.right (img < 3*thres,res);
11     SEND.down  (img < 4*thres,res);
12     SEND.down_r(img < 2*thres,res);
13   END;
14   RETURN res;
15 END dither_ordered;
```

The same approach can also be applied for larger patterns, e.g. 3×3 or 4×4. However, the complex algorithm

of Floyd-Steinberg error diffusion [5] is very hard to implement in data parallel, because of its inherent sequential nature.
Figure 7 shows the result of applying 2×2 *ordered dithering* to a grayscale image in comparison to simple thresholding.

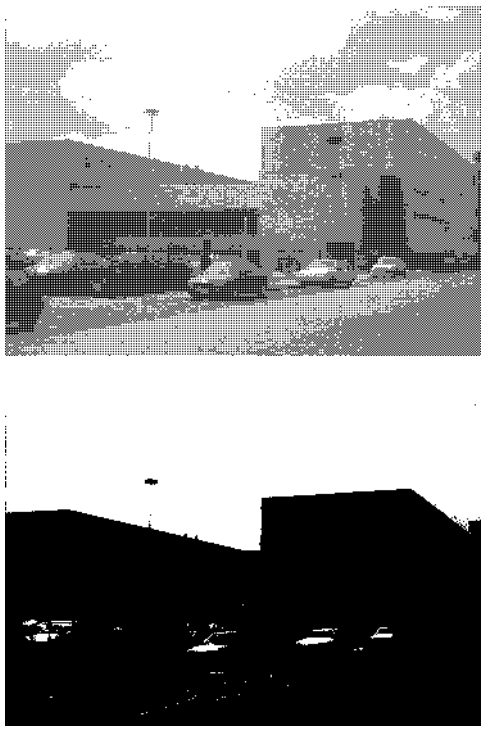

Figure 7: Ordered Dithering versus simple thresholding

3.4 Edge Detection

Edge detection is a central task in low level image processing. Edge points are characterized by a high local difference *(gradient)* in gray values. Edge strength and direction may be used as features for subsequent image processing. The idea behind this approach is that edges usually (but by no means always) represent the outline of objects in an image. Unfortunately, the reverse also isn't true in general. An alternate approach to object recognition is area-based image segmentation, which will be discussed later. Well known edge filters are Robert's Cross, Laplace, Sobel, Kirsch, Prewitt, and Marr-Hildreth [1], [7], [10]. Here, we will discuss a data parallel version of the Sobel operator.
The Sobel operator comprises two filters: one for detecting horizontal edges and one for detecting vertical edges (Figure 8). Combining both values in the same way as transforming Cartesian coordinates to Polar coordinates results in strength and direction of edges.
Our data parallel implementation of the Sobel filters in *x*- and *y*-direction again makes use of the separability of these filters. The naive implementation of Sobel-x would be:

```
RETURN   MOVE.down_l(img)
       + 2 * MOVE.left(img) + MOVE.up_l(img)
       - MOVE.down_r(img)
       - 2 * MOVE.right(img) - MOVE.up_r(img);
```

–1		1
–2		2
–1		1

Sobel-x

1	2	1
–1	–2	–1

Sobel-y

Figure 8: Sobel filter templates

This would require six data exchange operations in a eight-neighborhood. However, if we separate this two-dimensional filter into two single-dimensional filters, we will need only four data exchange operations in a four-neighborhood. Figure 9 demonstrates this filter separation by applying matrix multiplication. This approach can be used for any local filter, provided that the filter matrix can in fact be expressed as the product of two vectors.

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

Figure 9: Separation of Sobel template

Program 7: Sobel operator in x- and y-dimensions

```
1  PROCEDURE sobel_x_3x3(img:
2        grid OF gray): grid OF INTEGER;
3  VAR col: grid OF INTEGER;
4  BEGIN
5    col := 2*img + MOVE.up(img)
6                 + MOVE.down(img);
7    RETURN   MOVE.left(col)
8           - MOVE.right(col);
9  END sobel_x_3x3;
```

```
1  PROCEDURE sobel_y_3x3(img:
2        grid OF gray): grid OF INTEGER;
3  VAR row: grid OF INTEGER;
4  BEGIN
5    row := 2*img + MOVE.left(img)
6                 + MOVE.right(img);
7    RETURN   MOVE.down(row)
8           - MOVE.up(row);
9  END sobel_y_3x3;
```

Program 7 shows the implementation of Sobel-x and Sobel-y, Figure 10 shows the results. It is clear to see that one filter only recognizes vertical edges, while the other one recognizes only horizontal edges.
The already mentioned transformation of *x*- and *y*-edges to Polar coordinates' strength and direction is:

$$b = \sqrt{dx^2 + dy^2} \qquad r = \operatorname{atan}\frac{dy}{dx}$$

The data parallel implementation of these formulas is identical to the sequential implementation of a *single pixel*, since this operation is required for every pixel in the

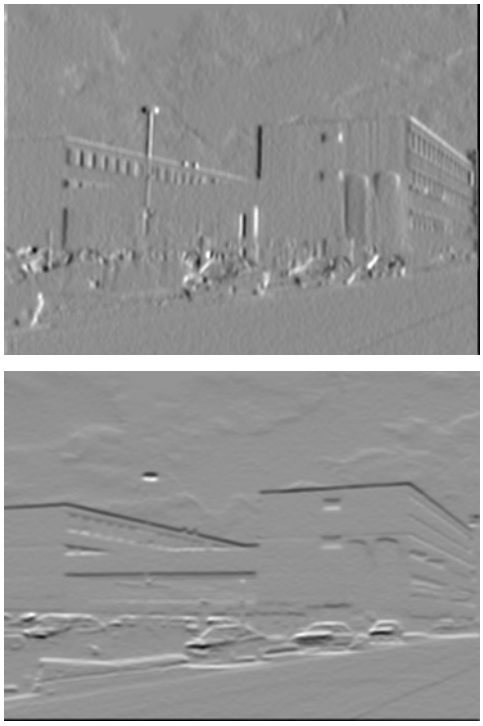

Figure 10: Vertical and horizontal edges

whole image and can be executed in parallel without restrictions. In Program 8, the Sobel-x and Sobel-y filters are called as subroutines. Despite using the specified square and square-root functions, the program has been simplified to use the sum of the absolute values as a faster approximation. Also, the edge direction has been reduced to 256 integer values.

Program 8: Sobel edge detection

```
 1 PROCEDURE edges_sobel_3x3(
 2           img: grid OF gray;
 3           VAR  strength,direction:
 4                  grid OF gray);
 5 (* edge strength and direction *)
 6 VAR dx,dy: grid OF INTEGER;
 7 BEGIN
 8   dx := sobel_x_3x3(img);
 9   dy := sobel_y_3x3(img);
10   strength := limit2gray(
11                 ABS(dx) + ABS(dy) );
12   direction:= round((arctan2(
13               FLOAT(dy),FLOAT(dx))
14             + pi) / (2.0*pi)*255.0 );
15 END edges_sobel_3x3;
```

Figure 11 now shows the final result of the Sobel edge detection. Next to the original image is the edge strength; stronger edges are represented by darker shades of gray. In the lower left, thresholding has been applied to the edge strength, leaving a binary image, e.g. for subsequent application of morphologic operators (see below). Finally, lower right shows the edge direction with the angular

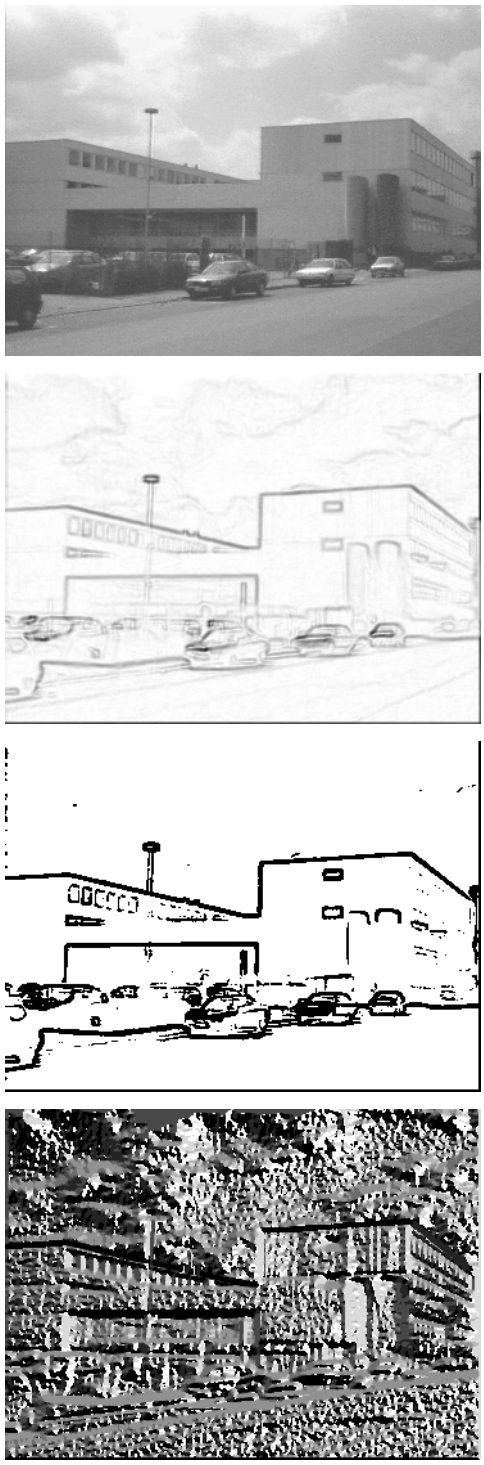

Figure 11: Original, edge strength, thresholding of edge strength, edge direction

4. Morphologic Operators

Morphology is the science of form, gestalt, and organization, while the term originates from biology. In image processing, morphologic operators operate on the outer form of an object. While there are morphologic operators for grayscale images [8], we will only discuss binary morphologic operators.

4.1 Erosion and Dilation

The two basic morphological operators are *erosion* and *dilation*. *Erosion* subtracts pixels from the border of an object (objects are black = `TRUE`, background is white pixels = `FALSE`), while *dilation* adds pixels to the outline of an object. Both operators use a *structure element* as a parameter, in order to find particular pixel patterns in an image. Overlaying the original image with the structure element, all black entries in the structure element have to match with black pixels in the neighborhood of the pixel examined, in order to produce a match. White entries in the structure element have no meaning (*don't care*), they do *not* require white pixels in the neighborhood.

For binary images, the structure element is a binary pattern of the filter size, e.g. a 3×3 matrix of 0s and 1s. The simplest structure element therefore has nine back pixels (1s) and no white pixel (0s) (see S_1 below). Another example is S_2, with black pixels along the diagonal and white pixels elsewhere. Structure element S_3 can be used to represent four-neighborhoods.

$$S_1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad S_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad S_3 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Erosion: If *all* black elements in the structure element match the neighbors of a pixel, the original pixel becomes black, if not it becomes white.

Dilation: If *at least one* black element in the structure element corresponds to a neighbor pixel, the original pixel becomes black, if not it becomes white.

Using another definition, *erosion* and *dilation* can be found by shifting the whole image and using pixel-wise set intersection (AND) and set union (OR). This approach is especially suited for data parallel processing. In the following definition, *A* denotes the original image, *S* is the structure element.

$$\text{Erosion} \ominus \qquad A - S = \bigcap_{v \in S} MOVE \bullet v(A)$$

$$\text{Dilation} \qquad A \oplus S = \bigcup_{v \in S} MOVE \bullet v(A)$$

For symmetric structure elements, *erosion* and *dilation* are again separable local operators.

Figure 12 shows the application of both operators with structure element S_1 (3×3 `TRUE`). *Erosion* almost completely deletes the object in the foreground and only leaves object pixels (black) completely surrounded by other object pixels. *Dilation* enlarges the object in foreground by adding more black pixels to it.

Program 9 shows the data parallel implementation of *erosion* and *dilation*. Here, the structure element S_1 has been implicitly assumed, instead of the more general approach of passing it as a parameter. Calculation starts within lines, then continues within columns.

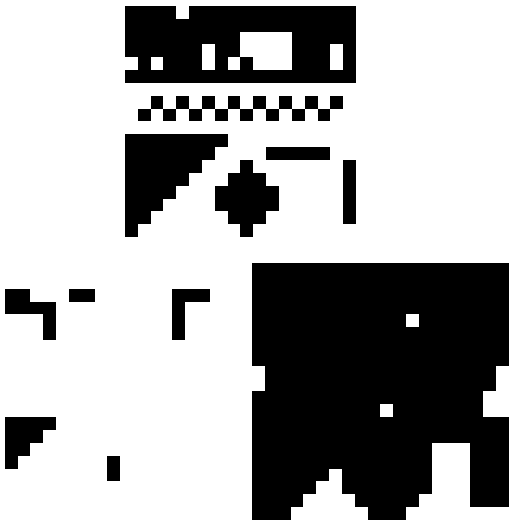

Figure 12: Original, erosion, and dilation

Program 9: Erosion and Dilation

```
 1 PROCEDURE erosion(
 2           img: grid OF binary):
 3           grid OF binary;
 4 VAR res: grid OF binary;
 5 BEGIN
 6   res := img AND MOVE.left(img)
 7              AND MOVE.right(img);
 8   res := res AND MOVE.up(res)
 9              AND MOVE.down(res);
10   RETURN res
11 END erosion;
```

```
 1 PROCEDURE dilation(
 2           img: grid OF binary):
 3           grid OF binary;
 4 VAR res: grid OF binary;
 5 BEGIN
 6   res := img OR MOVE.left(img)
 7              OR MOVE.right(img);
 8   res := res OR MOVE.up(res)
 9              OR MOVE.down(res);
10   RETURN res
11 END dilation;
```

4.2 Open and Close

Morphologic operators *open* and *close* are directly based on *erosion* and *dilation*. They are simple sequences of the two (again, *A* is original image, *S* is structure element):

$$\text{Open} \qquad A \circ S = (A \ominus S) \oplus S$$

$$\text{Close} \qquad A \bullet S = (A \oplus S) \ominus S$$

Operation *open* deletes "fuzzy" boundaries of an object, while *close* can be used to close small gaps within an object. Program 10 shows the most simple implementation of subsequent execution, while Figure 13 demonstrates the effect of applying *open* and *close* to the same input image as before.

Program 10: Open and Close

```
1  PROCEDURE open(img:
2     grid OF binary): grid OF binary;
3  BEGIN
4    RETURN dilation( erosion(img) )
5  END open;
```

```
1  PROCEDURE close(img:
2     grid OF binary): grid OF binary;
3  BEGIN
4    RETURN erosion( dilation(img) )
5  END close;
```

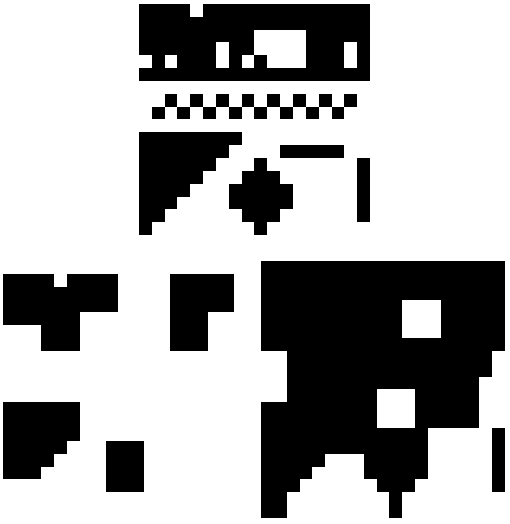

Figure 13: Original, open und close

Figure 14 demonstrates the results of morphologic operators *erosion*, *dilation*, *open*, and *close* on a larger natural input image, which is the previously computed edge strength threshold of the Sobel operator.

4.3 Fill and Connected

Further morphologic operators are the iterative filling of object area or background area. This can be done by using the operators *fill* and *connected*. Both operators need a starting position within the image. *Connected* iteratively uses the operation *dilation* intersected with the image foreground (objects = black pixels) until no more changes occur. *Fill* works similar, but uses intersection with the background instead of the foreground.

Operation *fill* fills all free space within an object (according to the structure element), while operation *connected* returns the connected component from the given starting position.

The following iteration definition holds for *fill* applied to image A:

$fill_0$:= specified starting point (usually background, i.e. `FALSE`)

$fill_k$:= $dilation(fill_{k-1}) \cap \overline{A}$

Transforming this definition into a data parallel program is simple (Program 11). The iteration is implemented as a

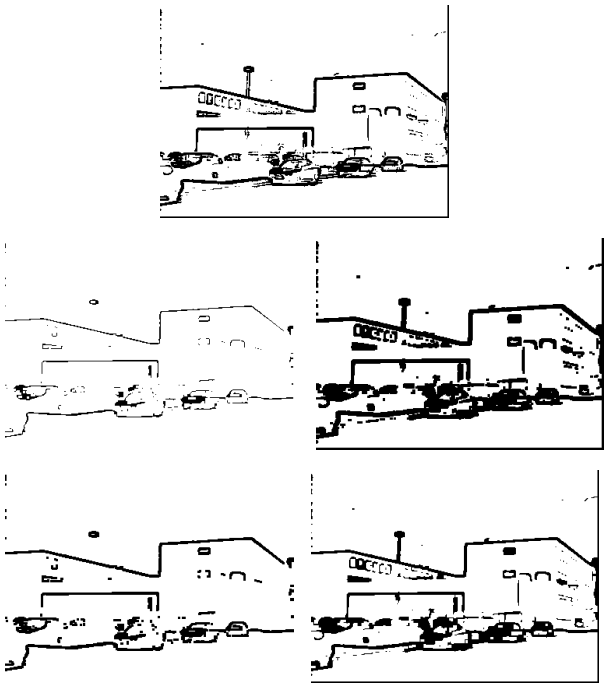

Figure 14: Original, erosion, dilation, open and close

REPEAT-loop, while the termination condition has to check whether no pixel at all has changed during the last iteration. This is done by using reduction with AND, comparing old and new image pixel-wise in parallel. Care has to be taken, to assure that the loop will terminate. As it is easy to see, the new image is generated by applying *dilation* and intersection (AND with the *complement* of the original image) to the old image. Therefore, when starting from the background, each iteration may add additional black pixels, but will never take existing black pixels away. This guarantees the termination of the loop after a finite number of steps.

Program 11: Fill

```
 1  PROCEDURE fill(img: grid OF binary;
 2            start_x,start_y: INTEGER):
 3            grid OF binary;
 4  VAR grow, new_grow, not_img:
 5            grid OF binary;
 6  BEGIN  (* init grow, start is TRUE *)
 7    new_grow := (DIM(grid,1) = start_x)
 8              AND (DIM(grid,2)= start_y);
 9    not_img  := NOT img;
10    REPEAT
11      grow := new_grow;
12      new_grow := dilation(grow)
13                  AND not_img;
14    UNTIL REDUCE.AND(grow = new_grow);
15    RETURN grow
16  END fill;
```

The following iteration definition holds for *connected* applied to image A:

con_0 := specified starting point
(usually foreground, i.e. TRUE)
con_k := $dilation(con_{k-1}) \cap A$

So the only difference to operator *fill* is the fact that here the image is intersected with the original image instead of the complement. Program 12 shows the data parallel implementation of operator *connected*.

Program 12: Connected

```
 1 PROCEDURE connected(img:
 2           grid OF binary;
 3           start_x,start_y: INTEGER):
 4           grid OF binary;
 5 VAR grow, new_grow, not_img:
 6           grid OF binary;
 7 BEGIN  (* init grow, start is TRUE *)
 8   new_grow := (DIM(grid,1) = start_x)
 9           AND (DIM(grid,2) = start_y);
10   REPEAT
11     grow := new_grow;
12     new_grow:=dilation(grow) AND img;
13   UNTIL REDUCE.AND(grow = new_grow);
14   RETURN grow
15 END connected;
```

Figure 15 shows operators *fill* started at (12,5) and *connected* started at (3,3) at an example. *Fill* fills an empty space inside the top black object. Note the outside pixel, only connected in a single point. This is a consequence of the previously chosen structure element S_1 (eight-neighborhood). Using structure element S_3 instead (four-neighborhood), the operator would have stopped without this extra pixel. Operator *connected* returned the lower left triangle, which is clearly separated from the white background.

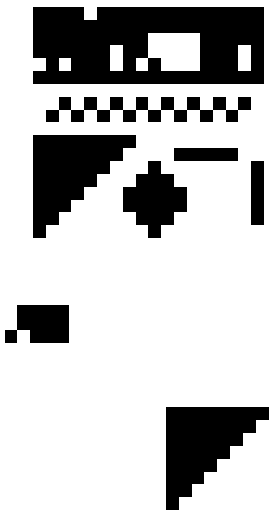

Figure 15: Original, fill(12,5), connected(3,14)

4.4 Boundary and Skeleton

The morphological operation *boundary* is a very efficient method to find the outline of an object. It is computed by forming the difference between original image and its *erosion*:

$$\text{Boundary}(A) = A - (A \ominus S)$$

Program 13 displays *boundary's* direct parallel implementation.

Program 13: Boundary

```
1 PROCEDURE boundary(img:
2       grid OF binary): grid OF binary;
3 BEGIN
4   RETURN img AND NOT erosion(img)
5 END boundary;
```

One possible definition of the skeleton operator has been proposed in [7]:

$$\text{Skeleton}(A) = \bigcup skel(A)$$

$$= \bigcup \{(A \ominus kS) - [(A \ominus kS) \circ S]\}$$

Here, kS denotes k-times applying an operation (here: *erosion*) with structure element S. The Union (OR function) is computed in an iteration until the iterated *erosion* of A results in an empty (all white) image.

Program 14: Skeleton

```
 1 PROCEDURE skeleton(img:
 2     grid OF binary): grid OF binary;
 3 VAR skel, k_times: grid OF binary;
 4     ready: BOOLEAN;
 5 BEGIN
 6   skel := b_white;
 7   k_times := img;
 8   WHILE REDUCE.OR(k_times) DO
 9     skel := skel OR
10      (k_times AND NOT open(k_times));
11     k_times := erosion(k_times);
12   END;
13   RETURN skel
14 END skeleton;
```

Program 14 shows the data parallel implementation of this version of *skeleton*. Starting with an empty (all white) image, the OR function computes the union operation in parallel within the WHILE loop. The loop continues while there is at least one pixel left. This condition is again computed in parallel by using reduction. One variable is used to *"erose-down"* the original image to an empty (all white image), while another variable is used to construct the resulting skeleton from an initially empty image. Figure 16 illustrates the operation of operators *boundary* and *skeleton* on the same input image as above.

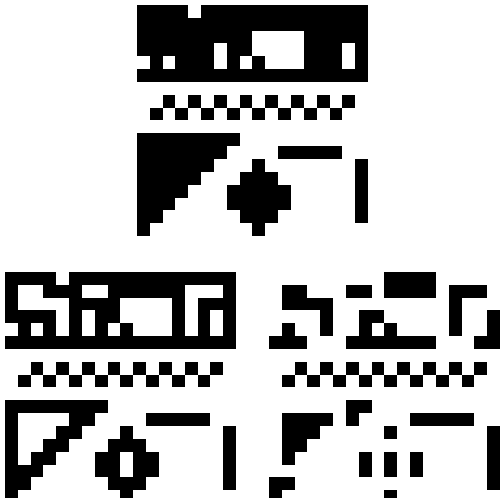

Figure 16: Original, Boundary, skeleton

5. Region-based Image Operators

All image operators in the previous sections worked with single points or within a limited neighborhood. Now, we want to take a look at region-based operators. An important application in this context is the *segmentation* of an image, that is the partitioning of an image in a number of coherent pixel areas.

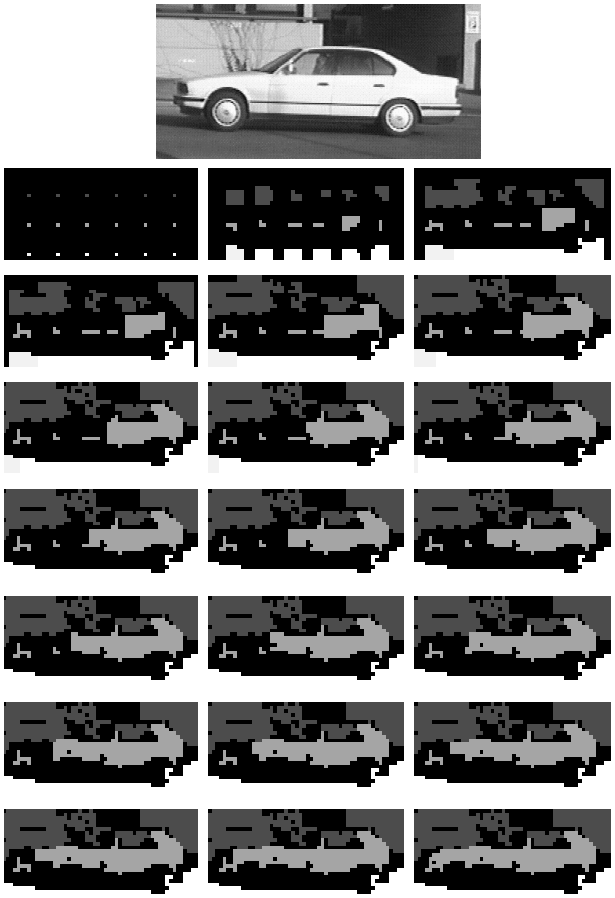

Figure 17: Stepwise execution of region growing

The following rules must hold for segmentation:

(a) $A = \bigcup R_i$ union of all regions covers the whole image

(b) R_i is coherent

(c) $R_i \cap R_j = \varnothing$ regions do not overlap

(d) $P(R_i) = \texttt{TRUE}$ pixels within a region fulfil uniformity condition

(e) $P(R_i \cup R_j) = \texttt{FALSE}$ union of two regions does not fulfil this condition

Here, predicate P defines the required uniformity of a region. It may involve gray values or other pixel or area features. A rather simply implementation is to define uniformity by a maximum gray value difference between pixels in a region.

In the following, we will transform a sequential approach to *region growing* from [6] into a data parallel program. The algorithm starts with a number of *seed points*, which usually are placed at fixed distances in the image. E.g. in Figure 17, every eighth row and column is chosen as a seed point, so this makes 1/64th of all pixels to seed points. Initially, all pixels belong to region zero (background).

An iteration follows, which terminates when no pixel changes its region in the last loop execution. In each iteration, each pixel sends its own gray value and region number to its four nearest neighbors. If the difference to a neighbor pixel's gray value is less than a specified threshold and the neighbor's region number is higher than the own one, then a pixel *migrates* to the neighbor's region. The latter condition is only required to guarantee loop termination. Otherwise a pixel may cyclically migrate to one or the other of two adjacent regions.

Figure 17 exemplifies the stepwise operation of *region growing*. The original image of a car is small enough to recognize individual pixels. Good to see are the seed points, which have been set in an 8×8 raster. From these, the regions spread over the image, while their corresponding grayscales are only used to distinguish regions – they do *not* correspond to the gray values of these pixels *(false color representation)*. After a few iterations, most of the image is already covered by regions, the final iterations contribute only minor changes between neighboring regions. The data parallel version of this algorithm is described in detail in [4].

6. Conclusion and Future Work

We have demonstrated the usefulness of synchronous parallel processing for real time image processing. We have examined a number of low-level routines and their data parallel implementation. Prominent application areas are active vision, especially for mobile robot systems, and general imaging systems, which can gain considerable speedup by employing SIMD sub-systems. Ideally, the use of a data parallel system would be transparent to the user, who is simply using an image processing library. The translation of an existing sequential image processing library to a parallel environment has to be done only once. This can be performed by experts to en-

sure optimal utilization of the parallel resources. The application programmer himself does not have to deal with parallel processing at all.
All presented parallel algorithms have been implemented in Parallaxis and can be executed either on a data parallel system or with the Parallaxis simulator on sequential workstations. Future work will be the design of an SIMD vision sub-system. Several different approaches are possible:

- Board Approach
  consisting of 128 signal processors, designed with an external ISA bus for PC integration
- Gate Array Approach
  designing a general purpose SIMD system with 2-dim. grid structure on a chip
- FPGA Approach
  designing and analyzing the required structure of a single ALU

Further details can be found in the research book [4], from which the images, formulas and programs of this publication have been taken.

7. Acknowledgments

The author would like to thank Stefan Feyrer, Wolfgang Rapf, and Michael Reinhardt for implementing various high level image processing routines, not discussed in this article. Also acknowledged is the implementation work of Hartmut Keller, Jörg Stippa and many others on the Parallaxis programming environment.

8. References

[1] Ballard, D. , Brown, C., *Computer Vision*, Prentice Hall, Englewood Cliffs NJ, 1982

[2] Bräunl, T., *Parallel Programming*, Prentice Hall, Englewood Cliffs NJ, 1993

[3] Bräunl, T., *Parallaxis-III – A Structured Data-Parallel Programming Language*, Proceedings of the First International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP-95, Brisbane Australia, April 1995, pp. 43–52 (10)

[4] Bräunl, T., with Feyrer, S., Rapf, W., Reinhardt, M., *Parallel Image Processing*, Springer Verlag, Heidelberg, 2000

[5] Foley, J., van Dam, A. , Feiner, S., Hughes, J., *Computer Graphics – Principles and Practice*, 2nd Ed., Addison-Wesley, Reading MA, 1990

[6] Gonzalez, R., Woods, R., *Digital Image Processing*, Addison-Wesley, Reading MA, 1992

[7] Jain, A., *Fundamentals of Digital Image Processing*, Prentice Hall, Information and System Sciences Series, Englewood Cliffs NJ, 1989

[8] Johansson, T., *Image Analysis Algorithms on General Purpose Parallel Architectures*, Ph.D. Thesis, Centre for Image Analysis, Rapport Nr. 16, Univ. Uppsala, Sweden, 1994

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

[10] Nalwa, V., *A Guided Tour of Computer Vision*, Addison-Wesley, Reading MA, 1993

[11] Thinking Machines Corporation, *C* Programming Guide Version 6.0*, Thinking Machines System Documentation, Nov. 1990