Implementing Image Processing Applications on a Real-Time Architecture

G. QUENOT and C. COUTELLE and J. SEROT and B. ZAVIDOVIQUE
16 bis Avenue Prieur de la Côte d'Or, 94114 Arcueil Cedex
E-mail: quenot@etca.fr, coutelle@etca.fr, jserot@etca.fr, zavido@etca.fr

Abstract **This paper presents three examples of real-time image processing applications that were implemented on a data-flow architecture developed at the ETCA. Low-level image processing is performed on a regular three-dimensional network of 1024 custom data-flow processors. Image features previously extracted in the low-level step are handled by a two-dimensional network of 12 general purpose processors. Fast prototyping of real-time image processing applications is achieved through a programming environment including a complete stream from functional programming specification to network configuration. A large class of algorithms can be implemented. Among them we describe a non-linear filter, a connected component labeling and a colored object tracking.**

Figure 1: The Functional Computer

I. Introduction

The algorithmic complexity and/or the huge amount of sensory data involved in many realistic image processing applications make computer vision highly computer-power demanding. It is especially true when a specific application requires video frames to be processed in real-time: fast moving object tracking, non-controlled environment watching are two examples of such applications. Computing powers in the range of billions operations per second are then commonly encountered. For such tasks, Von Neumann style or other classical architectures fail to provide sufficient performance at a reasonable cost. This is mainly due to the fact that inherent parallelism of many image processing algorithms cannot be exploited using these classical architectures. All the more as this parallelism often changes along with processing types. Indeed many image processing algorithms can be decomposed in low-, intermediate- or high-level tasks [12]. Examples in each of the three processing levels are:

- convolution, corners extraction or connected component labeling (low-level),
- right angles extraction in a line segment image or convex hull computation (intermediate level),
- model matching, assumption tree exploration and hypothesis validation (high-level).

Thus many innovatives architectures have been proposed as alternatives to Von Neumann ones, each of them being generally well-suited for a given computation type and a given data structure type. For example, it is well-known that SIMD (Single Instruction Multiple Data streams) architectures are particularly suited to low-level image processing where the two-dimensional structure of the image still prevails.

But most often such architectures are inefficiently exploited. It appears that the most powerful a machine is, the less programmable it becomes in the sense that the programmer should be an architecture specialist. It has been shown [1] that these difficulties could be overcome thanks to an appropriate formalism for parallelism expression at both algorithmic and architectural level. The proposed formalism is that of data-flow graphs (DFG). Vision tasks are decomposed into elementary actions per pixel corresponding to DFG nodes. Arcs denote elementary action dependencies. This functional decomposition is naturally expressed in a language of the functional programming [3] type [11] and its architectural counterpart is a massively parallel data-flow machine [9] [10] also called "Functional Computer" (see figure 1).

II. Hardware description and execution model

Low-level image processing generally involves a fixed set of elementary operations on image data structures. It is supported by a three-dimensional network of 1024 custom data-flow processors (DFP) that have been extensively described in [9] (see appendix D). Each DFP is able to perform up to 50 millions 8- or 16-bit operations per second. Due to the mesh connected structure of the processor network, only local interconnections are provided. DFP hardware has been designed in order to satisfy most of the basic operations encountered in low-level image processing: for example a hardwired histogrammer has been integrated in the DFP structure.

Intermediate-level processing is devoted to a two-dimensional network of 12 T800 Transputer modules each including 1 Mbyte memory. Transputers are general purpose processors, directly incorporating communication links.

Each processor (DFP or Transputer) operates according to data-flow principles ([4], [2]): in particular, output flows must only be functions of input flows independently of what may occur in other processors. This is achieved:

- in DFPs by a local dynamic firing rule validating operator execution if all needed data are available on input stacks and if output stacks are not full;
- in Transputers by a single parallel Transputer process reading input tokens and writing output tokens on logical input/output "slots" whithin an infinite loop.

Since the DFP network has to handle image data in real-time, its bandwidth is video (10-25 Mbytes per second), allowing all frames to be processed as soon as they are digitized. The Transputer link is limited to a 2 Mbyte per second bandwidth. A communication mechanism has therefore been elaborated to make cooperation possible between the two heterogeneous networks (this will be detailed in the following section).

As shown in figure 2, asynchronous flows are also supported. These asynchronous flows are delivered in and out via a low-bandwidth interface.

Each algorithm is executed through a gigantic pipeline typically involving more than one hundred simultaneous operations per step. Indeed a "functional parallelism" means all operations corresponding to one iteration are executed in parallel for only one video flow at every time step (Multiple Instruction Simple Data streams). This is to be compared for instance with "data parallelism", where only one operation is performed for all elements at every time step (Single Instruction Multiple Data streams). There is a one-to-one correspondence between physical processors and operations in the algorithm instead of a one-to-one correspondence between physical processors and data elements.

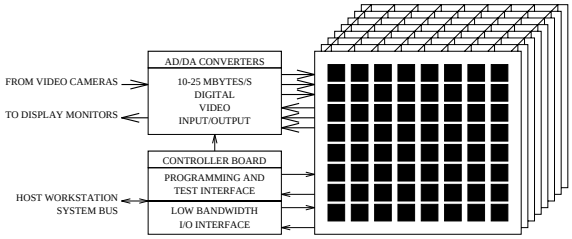


Figure 2: Hardware overview

The Functional Computer is connected to video cameras and display monitors through three input/output boards each of which is able to input/output one RGB color signal or three B/W signals.

III. Programming model and environment

According to our functional approach, a vision task is decomposed into elementary operators representing nodes of a directed data-flow graph. Arcs between nodes denote data dependencies. Execution of this DFG occurs as tokens flow along the arcs, into and out of nodes, according to a set of firing rules. These firing rules specify that a node may be active whenever tokens are present on its input arcs and tokens can be produced on its output arcs. When a node fires, input tokens are consumed, function results are computed and produced on the output arcs.

High-level programming of the Functional Computer is achieved through the use of a functional programming language [3] because of its nice properties for directed acyclic data-flow graph description [1]. The functional programming language available here to the end user exploits only a restricted dialect $< \mathcal{O}, \mathcal{P}, \mathcal{F} >$ of Backus's original functional language, in which:

- $\mathcal{O}$ is the set of objects which are all combinations of two basic object types: Pixel (a numerical value) and Control ("<" as $StartOfList$ and ">" as $EndOfList$). Since our application field is image processing, several commonly manipulated composite objects are provided:
  - Line: a structured sequence of Pixels ($< 1, 2, 3 >$);
  - Frame: a structured sequence of Lines ($<< 1, 2, >< 3, 4 >>$);
  - N-tuple: a fixed-length sequence of pixels $(1, 2, 3)$.
- $\mathcal{P}$ is the set of predefined functions or primitives. Algorithms are decomposed according to

the primitives available in the operator library. Operators can belong either to a DFP or to a Transputer or to a cluster of DFPs and/or Transputers (these composite operators are called macro-functions). The current version of the operator library contains about 150 elementary operators and 30 macro-functions. Its generality allows to dramatically improve the programmers efficiency since they do not have to redefine common primitives. For example, the convolution macro-function is provided because of its general applications in image processing (image smoothing, first or second order spatial derivatives approximations...). This is especially true for low-level operators which are implemented on the DFP network because a DFP code has to be assembled by a DFP architecture specialist.

- $\mathcal{F}$ is the set of functional forms (composition, selection, construction and insert forms are provided).

The network heterogeneity from DFPs and Transputers is obviously a matter of fundamental architectural differences. But from a data-flow point of view the main drawback is the incompatibility between bandwidths. Indeed data streams are usually flowing at an average speed in the video range (up to 25 Mbytes per second) inside the DFP network. Therefore Transputers cannot directly exchange data with DFPs because of their speed (2 Mbyte per second). A specific operator has thus been elaborated to overcome this limitation (denoted by \$ in functional programming sources). The internal structure of this operator is given in figure 3. It essentially consists of two coupled parts:

- an inner receiver which always accepts data from an outer emitter ;
- an inner emitter that delivers data as soon as the outer receiver is ready to accept them.

Outer emitter and receiver are either a DFP or a Transputer element, inner emitter and receiver are DFP processors. The \$ operator is also used when DFPs have to send or to accept data from the low bandwidth interface which is then considered as a slow receiver or emitter.

Finally, it should be noted that several distinct algorithms can be simultaneously mapped and independently executed. The only constraint is volume constraint: virtual functional processors are not implemented so the entire data-flow graphs have to fit in the physical network.

Figure 4 outlines the Functional Computer programming environment. As said in the introductive section, the DFG formalism fills in the gap between algorithmic and architectural domains. The programming environment therefore draws a complete stream from an algorithm functional description (DFFP source: Data-Flow Functional Programming source) to the network configuration loading.

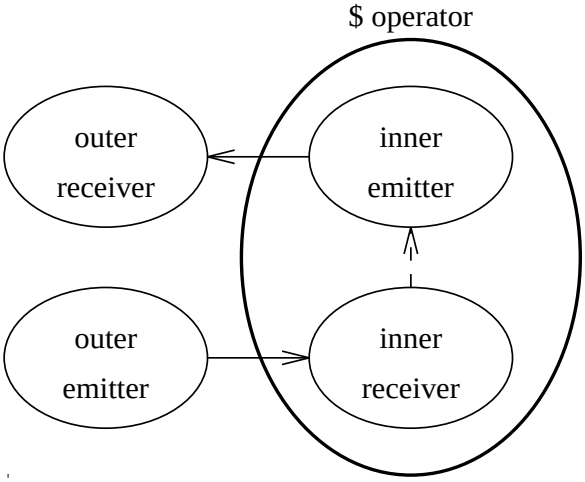


Figure 3: \$ coupling operator structure

IV. Non-linear filtering

When implementing image processing algorithms, it is often desirable to remove high frequency spatial noise. For example edge linking algorithms perform poorly under heavy noise conditions, resulting in a lot of gaps in the extracted contours. However, high frequencies are also embedded in features as contours.

This has led to the design of non linear filters known as "edge-preserving smoothing". Among them is the Nagao's [6]. It defines a set of 9 3x3 subwindows (8 directional subwindows and a central one) inside a 5x5 window as shown in figure 5. Then the current pixel (denoted by X in figure 5) is attributed the grey level mean value of the 3x3 subwindow with lowest variance.

A Nagao-like filter has been implemented that is based on the subwindow set shown in figure 6. It should be noted that each subwindow can be deduced from the lower right one by single translation thus making implementation simpler. Instead of the grey level mean the homogeneity factor computed in each 3x3 subwindow results from low-pass filtering and the disparity factor is the grey level maximum-minimum difference value instead of variance. The homogeneity factor of the 3x3 subwindow with the lowest max-min difference is then attributed to the 5x5 window central pixel.

A functional programming language description of our algorithm is given in appendix A, results are shown in appendix F.

The number of processors required for this algorithm

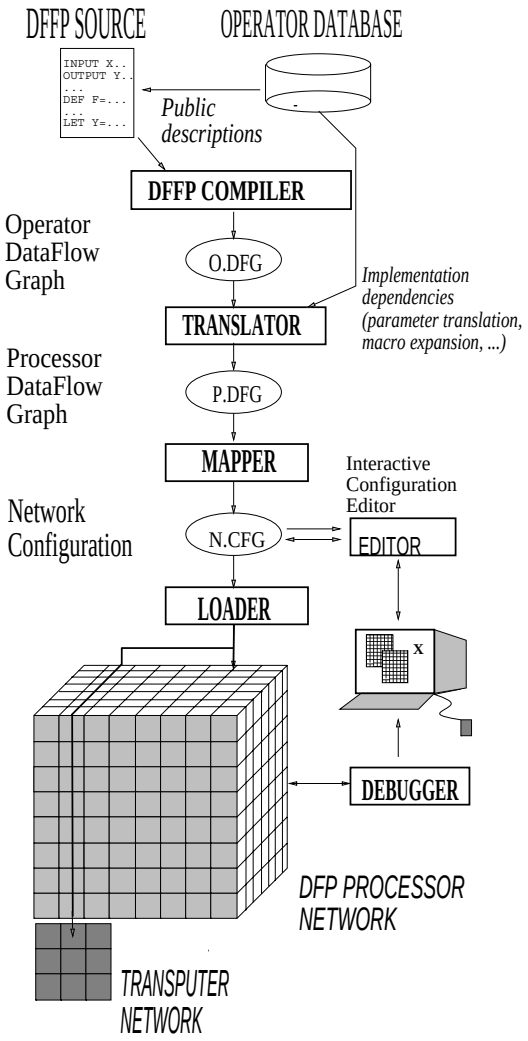


Figure 4: The Functional Computer programming environment

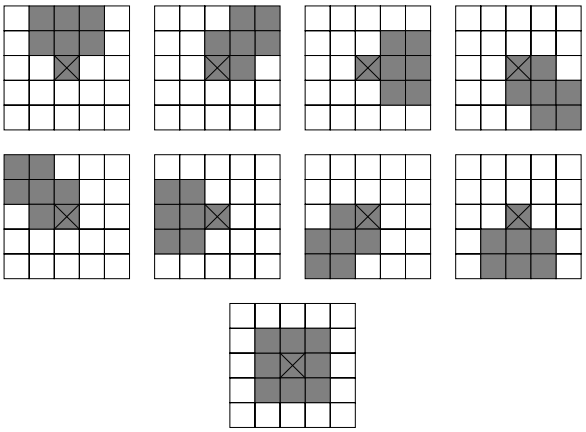


Figure 5: Nagao filter: subwindows

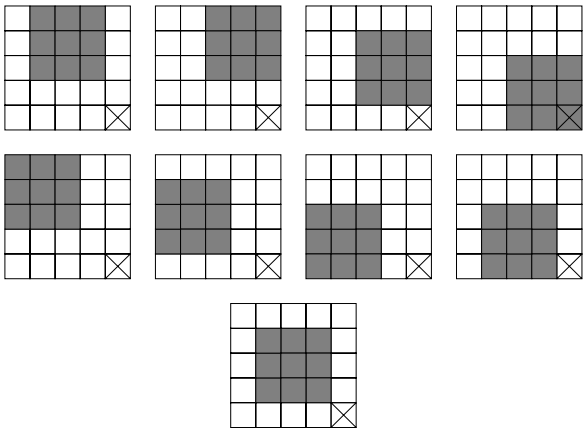


Figure 6: Nagao-like filter: subwindows

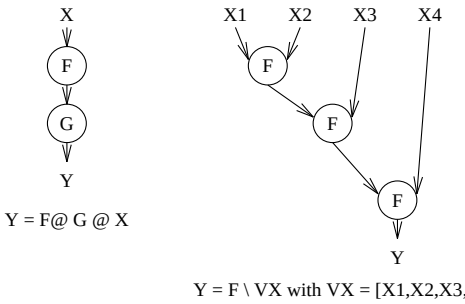


Figure 7: Composition and left-insert forms

is 141, including routing processors. Two functional forms are involved in the functional description: the composition form (denoted by @), and the left-insert form (denoted by \). The DFG associated with each of these two forms is shown in figure 7.

V. Connected component labeling

Connected component labeling is a fundamental operation in most intelligent vision systems. It consists in assigning a unique label to each connected region of a binary image. This algorithm allows a symbolic description of an observed scene through the computation of geometric moments, perimeter or gravity center coordinates for each region of the binary image. It is then possible to perform more complex tasks such as geometric model matching or other high-level jobs.

Because of the large amount of low-level operations attached to connected component labeling algorithms, real-time implementations are often based on dedicated hardware (see for example [7] and therein references). For the particular case of real-time operated pipeline architectures, the difficulty of implementing connected component labeling has been emphasized in [5]. It is mainly due to the recursive nature of such algorithms and to the fact that global informations

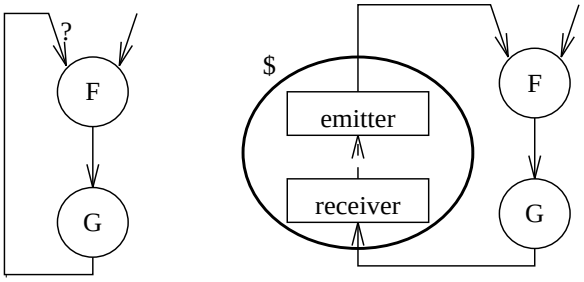


Figure 8: Recursive implementation with the $ operator

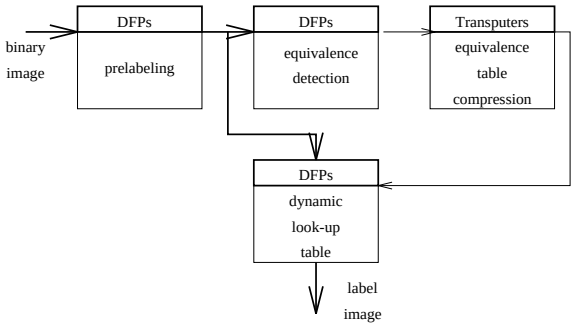


Figure 9: Connected component labeling algorithm outline

have to be handled. Inside the Functional Computer, recursivity may happen:

- at a DFP level; indeed each DFP datapath includes an ALU feedback (see [9]).
- at a Transputer level; recursivity is allowed by the C programming language (C function recursive calls).
- at a macro-function level; recursive operations are then implemented using the $ operator described in section III., as shown figure 8. According to data-flow principles, the F function cannot be executed until all needed input data are valid. So the $ inner emitter send an initial value to F in order to initialize the data feedback.

Our implementation of connected component labeling (fully described in [8]) relies on three steps (see also figure 9):

- the input binary image is first temporary labeled using a purely local operator;
- equivalent labels are detected and stored in an equivalence table;
- each equivalent label set is assigned a unique label and the temporary labeled image is transcoded using the new equivalence table.

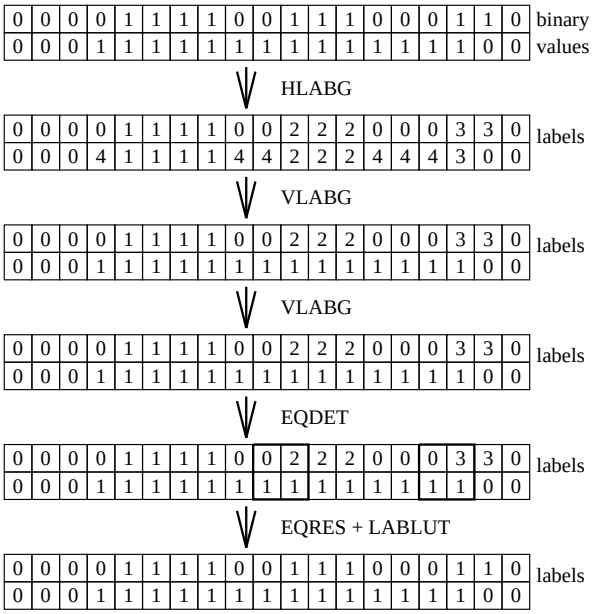


Figure 10: Labeling process sample

These three steps are frequently the core of scanline connected component labeling algorithms.

The associated functional programming source is shown in appendix B and comparative evaluation results in appendix E. Macro-function appearing in this description will now be explained.

- THRLD: digitized grey scale images are first interactively thresholded. The input threshold is manually set and read from the low-bandwidth interface.
- HLABG: each line is labeled independently of what may have occurred in upper lines. This is a pure horizontal information propagation. Only white pixels ("1" pixels) are labeled, other are unchanged ("0" pixels). HLABG thus labels horizontal line segments (the label dynamics is fixed to 16 bits allowing large image processing). It should be noted that HLABG is a recursive operator but this is no real problem since recursivity is here handled inside a single DFP;
- VLABG: this is the second function involving recursivity. This function aims at vertically propagating connectivity informations. If $B(i,j)$ is the (i,j) pixel binary value, $L_1(i,j)$ its "horizontal" label and $L_2(i,j)$ the VLABG function output, we have:

if $(B(i,j) = 1)$
  if $(B(i,j-1) = 0)$ then
    $L_2(i,j) = L_1(i,j)$
  else

$L_2(i,j) = L_1(i,j-1)$
else
$L_2(i,j) = 0$

But, as shown in figure 10, distinct labels can be assigned to the same segment. A given line segment is therefore attributed its lowest label by a recursive minimum operator.

- EQDET: equivalence detection between two labels corresponds to the situation where the same connected region has been attributed two distinct labels. In our algorithm, an equivalence is detected over a 2x2 neighborhood. Label $L(i,j)$ and label $L(i,j-1)$ are said to be equivalent whenever:

$$(B(i,j) = B(i,j-1)) and$$
$$(L(i,j) \neq L(i,j-1)) and$$
$$((B(i-1,j) = 0) or (B(i-1,j-1) = 0))$$

  This heuristic choice has not been formally proved sufficient but experiments involving real images and synthetic images (with U or spiral shapes) showed its efficiency. Two equivalent declared labels are sent to the next function (EQRES).

- EQRES: this function holds the equivalence-table compression. Compression stands here for calculating a unique label for all the labels in the same connected region. This step is devoted to a single Transputer module because such a table compression implies random access to the equivalence table. The compression algorithm used in this step is inspired from the one described in [5].

- LABLUT: the compressed equivalence table is downloaded in a look-up table so that the temporary labeled image can be transcoded.

About 100 DFPs are involved in the three-dimensional mapping of this algorithm.

VI. Colored object tracking

The last algorithm we describe is a moving object tracking. It is based on the assumption that targets have at least one uniformly colored part.

The tracking high-level functional source is shown in appendix C and comparative evaluation results in appendix E.

The first step is target designation, this is an open loop step. A tracking window is thus positioned onto the target thanks to the workstation mouse (x,y) coordinates (piped in the DFP network using the low-bandwidth interface). MUX multiplexer then outputs mouse (x,y) coordinates (Mouse-sl = 0). The tracking window dimensions are 256x256 pixels, the whole image being a 768x572 pixels image.

All along the open loop step, object color features are computed by a module referenced as "feature extractor" (REFCO in the functional source). These color features are summarized in a (R_m, G_m, B_m) triple representing (R,G,B) values statistical mean values over a small window ("feature window") centered at the (x,y) mouse position.

Then a human operator triggers the automatic tracking by pressing any mouse button. When so doing, MUX multiplexer outputs (x,y) coordinates computed by the tracker (Mouse-sl = 1). As automatic tracking is initiated, color features flowing out from the feature extractor are disabled. The color features available to the tracker just before tracking loop closing are memorized inside the tracker and become the reference features.

The so-called "tracker" module is in charge of maintaining the target inside the tracking window. This is achieved by:

- first: computing some kind of a distance (L_1 in (R,G,B) space) between tracking window pixels (R,G,B) components and the (R_m, G_m, B_m) reference features ($d(R,G,B,R_m,G_m,B_m) = |R - R_m| + |G - G_m| + |B - B_m|$). Each distance value is then thresholded ("white" pixels are likely object pixel). All these operations are performed by the CANPX macro-function;
- second: binary pixels previously extracted are projected along vertical and horizontal directions (XPROJ and YPROJ macro-functions);
- third: a new tracking window position is computed by the TWDXY macro-function (instanciated by two Transputers). The new x position (resp. y position) is the x projection (resp.) y projection median value.

Recursivity is naturally implied in this algorithm since the MUX function needs both mouse coordinates and tracker output coordinates (which do not exist until the tracking loop has been closed). It is denoted by the "$ @ Tracker-xy" term in the functional source.

About 466 DFPs are involved in the three-dimensional mapping of this tracking application. One of the twelve needed logical DFP plans is shown in figure 11. This graph has been manually optimized so that the ratio of processing elements is about 46% of the total DFP number (this ratio value seems to be the highest achievable value in this case, referring to our experiments so far).

VII. Conclusion

According to our functional approach, vision algorithm inherent parallelism can be expressed using the data-flow graph formalism. Indeed this formalism

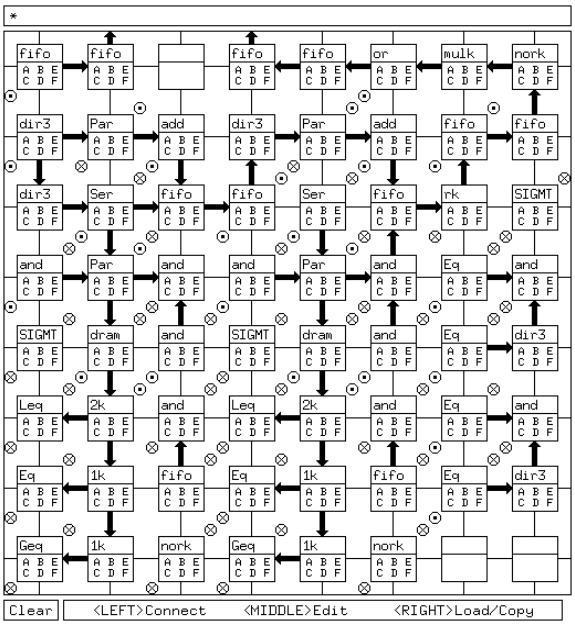


Figure 11: Part of a colored object tracking algorithm mapping (1 plane among 12)

seems to be adapted to both algorithmic and architectural descriptions, filling the gap between application programmers and architecture specialists.

From an end-user point of view, algorithms are decomposed into elementary actions represented by the primitive set of a functional language. They are physically implemented on a massively parallel data-flow machine comprising 1024 elementary data-flow processors for low-level image processing and 12 general purpose processors for intermediate level tasks. Three significant examples of image processing algorithms have been described and successfully implemented, contributing to validate our functional approach.

Future work is related with the automated derivation of vision automata from Functional Computer real-time emulation results. Our ultimate goal is indeed to increase efficiency of dedicated VLSI circuit design.

References

[1] E. ALLART and B. ZAVIDOVIQUE. Functional image processing through implementation of regular data-flow graphs. In *Twenty First Annual Asilomar Conference on Signal, Systems and Computers*, November 1987. Pacific Grove, CA.

[2] K.P. ARVIND and D.E. CULLER. Dataflow architectures. Technical Report MIT/LCS/TM-294, MIT Laboratory for Computer Science, MIT, Cambridge, MA, 1986.

[3] J. BACKUS. Can programming be liberated from von neumann style ? a functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.

[4] J.B. DENNIS. Data flow supercomputers. *Computer*, 13, November 1980.

[5] M. ECCHER. *Architecture parallèle dédiée à l'étude d'automates de vision en temps réel.* PhD thesis, Université de Franche-Comté, November 1992. In french language.

[6] M. NAGAO and T. MATSUYAMA. Edge preserving smoothing. *Computer Vision Graphics and Image Processing*, 9, 1979.

[7] C.J. NICOL. A systolic architecture for labeling the connected components of multi-valued images in real-time. In *Proceedings of the 1993 Conference on Computer Vision and Pattern Recognition*, pages 136–141, June 1993.

[8] S. PRAUD. Implantation d'un algorithme d'étiquetage en composantes connexes sur une machine flot de données dédiée au traitement d'images temps réel. Master's thesis, Université Paris-Sud, September 1993. In french language.

[9] G.M. QUÉNOT and B. ZAVIDOVIQUE. A data-flow processor for real-time low-level image processing. In *IEEE Custom Integrated Circuits Conference*, May 1991. San Diego, CA.

[10] G.M. QUÉNOT and B. ZAVIDOVIQUE. The etca massively parallel data-flow functional computer for real-time image processing. In *IEEE International Conference on Computer Design*, October 1992. Cambridge, MA.

[11] J. SEROT and G.M. QUÉNOT. Real-time image processing using functionnal programming on a data-flow architecture. In *Computer Architecture for Machine Perception*, December 1991. Paris, France.

[12] C.C. WEEMS and al. Architectural requirements of image understanding with respect to parallel processing. *IEEE Expert*, 6(5), October 1991.

A Nagao-like filter functional programming source

```
// Functional source implementing a Nagao-like non linear
filter


// Line pixel number
CONSTANT Np=768;
// Normalized low-pass filter coef.
CONSTANT Lp=1, 2, 1, 2, 4, 2, 1, 2, 1, 16


// MAIN -------
BEGIN MAIN [
VIDEO INPUT In :  FRAME(PIXEL);
VIDEO OUTPUT Out :  FRAME(PIXEL); ]
```

```
// 3x3 neighborhood extraction (input:  original frame)
DEF Vi = NEIG3 @ In;


// Disparity computation
DEF Ds = SUB @ [MAX  Vi, MIN  Vi];


// Low pass filtering (3x3 convolution using the Lp mask)
DEF Mn = CONV3(Lp) @ In;


// 3x3 neighborhood extraction (input:  disparity frame)
DEF Vd = NEIG3 @ Ds;


// 3x3 neighborhood extraction (input:  low pass filtered
frame)
DEF Vm = NEIG3 @ Mn;


// Partial associative sorting DEF Min1 = 1 @ Vd;
DEF Tmp1 = 1 @ Vm;
DEF Min2 = MIN @ [Min1, 2 @ Vd];
DEF Tmp2 = MUX @ [Tmp1, 2 @ Vm, EQU @ [Min2, 2 @ Vd]];
DEF Min3 = MIN @ [Min2, 3 @ Vd];
DEF Tmp3 = MUX @ [Tmp2, 3 @ Vm, EQU @ [Min3, 3 @ Vd]];
DEF Min4 = MIN @ [Min3, 4 @ Vd];
DEF Tmp4 = MUX @ [Tmp3, 4 @ Vm, EQU @ [Min4, 4 @ Vd]];
DEF Min5 = MIN @ [Min4, 5 @ Vd];
DEF Tmp5 = MUX @ [Tmp4, 5 @ Vm, EQU @ [Min5, 5 @ Vd]];
DEF Min6 = MIN @ [Min5, 6 @ Vd];
DEF Tmp6 = MUX @ [Tmp5, 6 @ Vm, EQU @ [Min6, 6 @ Vd]];
DEF Min7 = MIN @ [Min6, 7 @ Vd];
DEF Tmp7 = MUX @ [Tmp6, 7 @ Vm, EQU @ [Min7, 7 @ Vd]];
DEF Min8 = MIN @ [Min7, 8 @ Vd];
DEF Tmp8 = MUX @ [Tmp7, 8 @ Vm, EQU @ [Min8, 8 @ Vd]];
DEF Min9 = MIN @ [Min8, 9 @ Vd];
DEF Tmp9 = MUX @ [Tmp8, 9 @ Vm, EQU @ [Min9, 9 @ Vd]];


LET Out = Tmp9;


END
```

B Connected component labeling functional programming source

```
// Connected component labeling functional source


// Line pixel number
CONSTANT Np=768;


// MAIN -------
BEGIN MAIN [
VIDEO INPUT In:  FRAME(PIXEL);
VIDEO OUTPUT Out :  FRAME(PIXEL);
ASYNC INPUT Thr :  PIXEL; ]


// Input 8-bit coded grey level image thresholding
DEF Bin= THRLD @ [In, $ @ Thr];


// Binary image horizontal labeling
DEF Lbh = HLABG @ Bin;


// Label propagation
DEF Lbv = VLABG @ [Lbh, Bin];


// Equivalent label detection
DEF Edl = EQDET @ [Bin, Lbv];


// Label equivalence resolution
DEF Ers = EQRES @ Lbv;


// Dynamic label look-up table loading and label image
transcoding
DEF Lbl = LABLUT @ [Lbv, Ers];


// Label image outputting
LET Out = Lbl;


END
```

C Colored object tracking functional programming source

```
// Functional source implementing a colored object tracking.


// Line pixel number.
CONSTANT NP=768;


// MAIN -------
BEGIN MAIN [
VIDEO INPUT In :  FRAME(PIXEL);
VIDEO OUTPUT Out :  FRAME(PIXEL);
ASYNC INPUT Mouse-xy :  T-UPLE(2);
ASYNC INPUT Mouse-sl :  PIXEL; ]


// Tracking window coordinates selection (depending on Mouse-sl
value):
// - mouse coordinates when in open loop (Mouse-xy);
// - tracker coordinates when in closed loop (Tracker-xy).
// Recursivity lies in the ``$ @ Tracker-xy'' expression.  DEF
Twind-xy = MUX @ [$ @ Mouse-xy, $ @ Tracker-xy, $ @ Mouse-sl];


// Tracking window extraction.
DEF Tframe = TWIND @ [In, Twind-xy];


// Feature window extraction.
DEF Fframe = FWIND @ [In, Twind-xy];


// Reference color feature extraction
// and validation (by Mouse-sl value).
DEF Color = REFCO @ GATE @ [Fframe, Mouse-sl];


// Candidate pixel for tracked object extraction.
DEF Object-pl = CANPX @ [Tframe, Color];


// Candidate pixel for tracked object vertical and horizontal
projection.
DEF Object-px = XPROJ @ Object-pl;
DEF Object-py = YPROJ @ Object-pl;


// Tracking window coordinates computation.
DEF Tracker-xy = TWDXY @ [Object-px, Object-py];


END
```

D Reminder: processor specifities

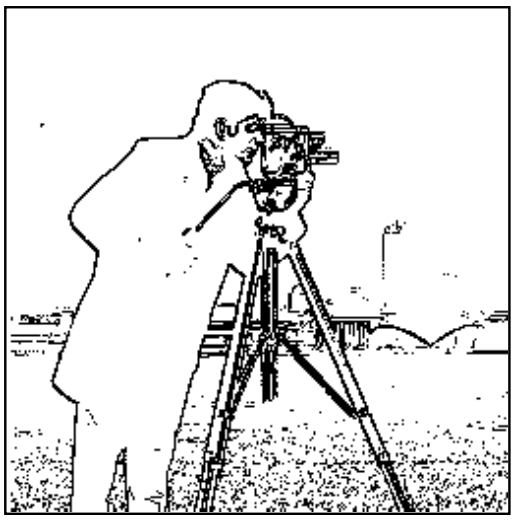
DFP chip main features	
Technology	CMOS $1\mu m$
Die size	$350mils \times 380mils$
Transistor count	160,000
Package	144 pins PGA
Operating frequency	25 Mhz
Integrated processors	2
Program memory	64×32 per processor
Data memory	256×9 per processor
Operations per second	up to 50 M per processor

E Comparative "evaluation" results

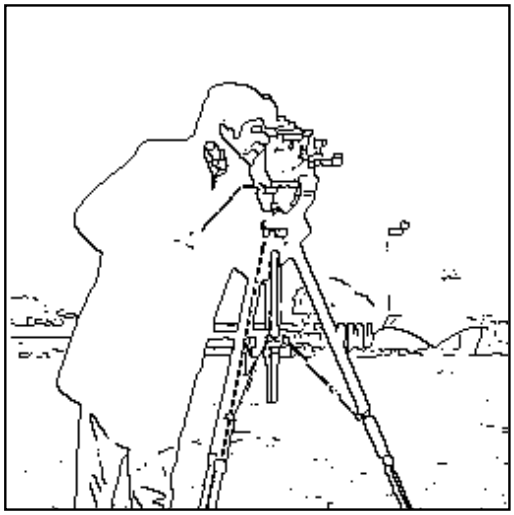
Application	Nagao-like	Labeling	Tracking
DFP number	141	100	466
Filling rate	57%	42%	46%
Operations per pixel	63	20	150
Computing time	$6\mu s$	40 ms	40 ms

Only the prelabeling is concerned here, that is independent of picture contents. Transputer run times in labeling and tracking are not considered here. From a strict application point of view, they are shorter than interframes anyway.

F Non-linear filtering results



Edge detection without preliminary Nagao-like filtering



Edge detection with preliminary Nagao-like filtering