Adding Associative Meshes to the PACCO I.P. Environment

Alberto Biancardi¹ and Alain Mrigot²

¹ Universit di Pavia, DIS, Via Ferrata 1, I-27100 Pavia, Italy

² Universit de Paris-Sud, IEF, Btiment 220,
F-91405 Orsay Cedex, France

Abstract. The data-parallel programming model, while perfectly suited for image processing, needs to be enhanced when shifting to image analysis. Associative meshes present two valuable benefits: they are susceptible of fast hardware implementations and they can express image computations based on connected components as whole image transformations. This paper presents, by means of coding examples, how some image analysis tasks can be expressed when a data parallel programming environment is aptly extended.

1 Introduction

A well know way of coding image processing algorithms with a good level of abstraction is to express pixel changes as whole image transformations. Such an approach is called data parallelism.

However, while this approach has proved to be straightforward when considering both pointwise and local operations, very few proposals have been done to incorporate higher level algorithms like region-based processing, shape analysis, etc. Indeed these algorithms exhibit much less regular data movements; hence their embedding as whole image transformations is more difficult.

On the other hand it turns out that most of these high level image analysis tasks may be expressed in terms of operations on connected sets of pixels, which can represent a region, a contour, or other connected parts with respect to the image topology. In the *associative net* model [7] these sets are represented by the connected components of so-called *mgraphs* and manipulated by primitives, performing global operations on images, called *associations* as described in section 2.

The associative net concept has been introduced in the Pacco image analysis programming framework, described in section 3, and the resulting extensions are presented in section 4. Section 5 presents a number of examples, covering some common image analysis tasks, which show the effectiveness of the approach.

2 Associative Nets

Given an image (of $\mathcal{N} \times \mathcal{M}$ pixels), the concept of neighbouring pixels can be formally described using the image planar graph representation [11] where

vertices represent pixels and edges represent the belonging of pixels to each other neighbourhood.

The associative mesh model stresses the fact that the neighbourhood of a pixel may depend on image data and therefore change from pixel to pixel. The ability to describe this locality is formalized by the *mgraphs*, i.e. pairs $g = (\mathcal{P}, \mathcal{E})$ where

$$\mathcal{P} = \{P_{xy} : xy \in [0, \mathcal{N}[\times [0, \mathcal{M}[\} \quad (1)$$

is the set of image pixels and

$$\mathcal{E} = \{\varepsilon_{xy \leftarrow vw} : xy, vw \in [0, \mathcal{N}[\times [0, \mathcal{M}[\} \quad (2)$$

is the set of graph edges indicating that pixel P_{xy} can gather information from pixel P_{vw} . $\mathcal{E}$ is a subset of the whole image graph. Neighbouring pixels whose data is accessed with respect to the mgraph g are defined ascendants:

$$\Gamma_g(xy) = \{P_{vw} : vw \in [0, \mathcal{N}[\times [0, \mathcal{M}[\text{ and } \varepsilon_{xy \leftarrow vw} \in \mathcal{E}\} \quad (3)$$

Iterated application of Γ_g defines the ancestors:

$$\hat{\Gamma}_g(xy) = P_{xy} \cup \Gamma_g(xy) \cup \Gamma_g^2(xy) \cup \dots \cup \Gamma_g^{\mathcal{N} \times \mathcal{M}}(xy) \quad (4)$$

The relation γ_g defined by

$$P_{xy} \gamma_g P_{vw} \text{ iff } \exists ij \in [0, \mathcal{N}[\times [0, \mathcal{M}[: P_{xy}, P_{vw} \in \hat{\Gamma}_g(ij) \quad (5)$$

is an equivalence relation where equivalence classes are the connected components of $\mathcal{P}$ with respect to g .

The relevance of mgraphs to image processing is that they can be used to control the evaluation of image operators by means of so called *associations*. Given an associative and commutative operator $\otimes$, an mgraph g , and an image P , the $\otimes$ -association(g, P) is a function returning an image A defined as

$$\otimes\text{-association}(g, P) = \left\{ A_{xy} = \bigotimes_{P_{vw} \in \hat{\Gamma}_g(xy)} P_{vw}, xy \in [0, \mathcal{N}[\times [0, \mathcal{M}[\right\} \quad (6)$$

Mgraphs can be considered a generalization of the concept of masks in recursive near-neighbourhood morphological operations in that they can define the regions within which information is propagated.

Just like morphological propagations [1,10], communication on mgraphs leads to asynchronous realizations that fully exploit the theoretical bandwidth. A chip to verify the feasibility of these machines has been developed and is under testing at the Institut d'Electronique Fondamentale of the University of Paris Sud [3].

As associations compute their results connected component per connected component, it is now clear the importance of mgraphs: they can be seen as parallel actions where the considered objects are no longer single pixels, but arbitrary sets of connected pixels. On the one side they embody a convenient tool to describe operations working on image regions, on the other side they offer the possibility to minimize the distance between the theoretical model and the actual hardware implementation.

3 The Programming Environment

Pacco [2] is an extension of a general purpose command language (Tcl[8]) and its graphical toolkit (Tk[8]). Its design is characterized by a *two-language* approach to object-oriented programming where flexible data-structuring and run-time extensibility let the programmer easily code highly interactive *applets* (mono-task programs) in addition to more traditional processing scripts.

Each Pacco object, called a *box*, stores a number of components which may be data segments or other container components (Cboxes), thus allowing the creation of component trees. This macro-structure can be used to fold together heterogeneous data-clusters and to carry additional hints about existing relationships within the data.

At the Tcl level data within a component is referred to by the component name: all the processing operations, called *actions*, treat the component as a whole. It is only at the C level that programmers gain access to the inner structure of components. As with other data-parallel approaches the main benefit is that actions abstract from the implementation details; they can be focussed on the working domain of the component data-type and range from basic task to highly refined ones.

4 Connected Components in Pacco: the Mgrp class

Programming support of connected components fits nicely into Pacco object model and is realized by a library containing the **Mgrp** class and a number of actions for images and numerical vectors, which deal with connected components. The **Mgrp** class encodes all the connected components of an image by linking together those neighbouring pixels that hold the same value. The inner structure of a **Mgrp** embeds a set of trees, each one spanning a connected component. Each tree is arbitrarily rooted into one pixel of the respective connected component. Thanks to the tree encoding non-absorbent, associative, and commutative, operators are handled correctly.

There are four families of operations that can be performed on connected components; these families come from a double dichotomy: data movement operations are subdivided into centralizing movements, when data is gathered by root nodes, and propagating movements, when data is distributed from the root to the leaves of each component tree; pixel operations, on the other hand, may

operations	context	action prefix
grouping from leaves (w/ update)	centralize	cnt
reducing on root (w/o update)	reduce	rdc
propagating from root	propagate	ppg
pix ops on connected components	ongraph	ong
pix ops on root nodes	onroots	onr

Table 1. Families of **Mgrp**-related actions.

be performed only on root nodes or on all the pixels belonging to the tree representation. Table 1 explains the organization and naming conventions adopted for these four families of actions. Implementation on a single serial processor leaves room for a further optimization: centralizing data on a connected component does not require the updating of the whole component if the behaviour must not be the same as the hardware mesh; therefore the *reducing* actions update only the root node.

Other commands allows information to be extracted from all the connected components thanks to the *collect* action, which gathers the root node values and stores them into a 1-D vector, and it can be distributed to connected components by using the *spread* action, which changes the root node pixels according to the vector values.

By splitting the theoretical association into two steps allows the programmer to diverge computation from root nodes on images to 1D vectors, bringing back the result to images if needed. This may be handier, because it is easier to work on a vector than on a sparse set of pixels, and more effective, because it avoids spanning component trees anyway.

A Note About Compression

Mgrp components may be used as a way to store smart shape information. The current implementation uses offsets from the first pixel of images; graph-related actions, therefore, incur in a minimal computational overhead because they require only one extra addition per pixel accessed.

While this method is effective at run-time, it is scarcely compressible because the offsets are all different! Anyway, a second form is possible where the graph stores the displacements of the children with respect to the father instead of their absolute offsets. This transformation is very fast, being performed by a few operations in a single scan of the graph, limits the variability of the offset information to eight values (plus the base offsets of the root nodes), and it is compatible with the memory model of Pacco components.

A final improvement can be gained by switching to an external format where the root node offset is filed as a 4-byte integer, while each child is identified by half a byte determining its position in the 3x3 neighbourhood, thus avoiding any dependence on image width.

Table 2 shows the sizes of the various methods when applied to different kinds of documents as shown in Figure 1. The last column of the table show

	No of foreground pixels	No of components	Compressed Size Using Child Offsets	Compressed Size Using Child Directions	ITU-T.6
text-mix	155591	945	104246	68638	19092
drawing	181376	129	80399	52610	11440
big-regions	403018	28	48507	29853	3862

Table 2. Comparison of different methods storing connected component information.

Fig. 1. Thumb-nails of the images used for compression evaluation.

the file length of the images using *ITU-T Recommendation T.6 (ex CCITT4)* [6] compression, that is the bit pattern without any additional information. In this way it is possible to (sort of) quantify how much overhead the ability to work on connected components costs in terms of additional storage required.

5 Coding Fragments

Usage of the `Mgrp` class will be shown presenting some standard processing tasks. These will be in the form of logs of a Pacco session.

The common structure of interactive sessions goes through the following steps: objects are allocated, using the command `obox`; component monitoring widgets are set up; data processing commands are issued, their syntax being *component.name action arguments*.

Initialization

```
obox imgs {
    {Img-byte src} {Img- tmp} {Mgrp graph}
}
```

We start by creating the `imgs` box. It stores three components: the image `imgs.src`, which will hold the image to be processed, the image `imgs.tmp`, which will act as working areas, and the `Mgrp` component `imgs.graph`.

```
imgs.src import szTest.pgm.gz
```

A test image is read into the `imgs.src` component. It is a monochrome image characterized by an overall, wavy noise.

```
imgs.graph engraph imgs.src
```

`imgs.src` connected components, i.e. connected pixels sharing the same non-zero value, are encoded by `imgs.graph`. From now on `imgs.graph` can be used independently from `imgs.src` contents.

Fig. 2. Example of application of size thresholding.

Size Thresholding

This last command terminates the initialization stage of our sample session. We now proceed

```
imgs.tmp clear 1
```

Initialize `imgs.tmp` so that each pixel stores the value 1.

```
imgs.tmp cnt/add imgs.graph
```

Compute the area of each component by using a plus operation grouping from leaves. When centralizing, each father pixel will increment its value by the sum of its children values; hence roots will store the number of pixels belonging to their respective connected component.

```
imgs.tmp ppg/set imgs.graph
```

We propagate each root value to all the pixels of each components; all the components will be set to a single value equal to the size of the component.

By thresholding the image, we can remove all the components smaller (or bigger) than a certain number of pixels. This is useful, for instance, when removing isolated pixels whose presence is due to noise. The implementation of the Morphological Image Cleaning procedure [9] can be simplified by replacing the skeletonization/reconstruction step with a simple *size thresholding* of smallest components.

```
imgs.tmp threshold 10.5
```

This command removes all the components that have less than 10 pixels as shown in Figure 2. Thanks to size thresholding thin lines are preserved (while, of course, details smaller than 10 pixels are lost).

Centroids (and Higher Order Moments)

We now proceed showing how to compute the inertial moments of connected components; once moments are determined, moments invariants [5] are easily found.

```
set paccodefaults(Zen_:length) [imgs.graph count]
```

As we are going to create some 1-D vectors, we change the default length of `Zen_` components and make it equal to the number of connected components of `imgs.graph` so that we can omit its specification on `Zen_` creation.

```
obox param {
  {Zen_ area} {Zen_ mx} {Zen_ my} {Zen_ mxx}
  {Zen_ aux}
}
```

The box `param` will hold all the item parameters. It is made of numerical vectors (`Zen_` components) whose length, being unspecified, is equal to the default length (i.e. the number of connected-components in the test image).

```
imgs.tmp clear 1
imgs.tmp cnt/add imgs.graph
param.area collect imgs.graph imgs.tmp
```

After computing the size of each component by summing up the number of their pixels, the elements of the vector `param.area` receive the values of image `imgs.tmp` stored in the root pixels according to `imgs.graph`.

After evaluating areas, we can compute first order moments.

```
imgs.tmp clear -x
  Set each pixel of image imgs.tmp to its  $x$  coordinate.
```

```
imgs.tmp cnt/add imgs.graph
param.mx collect imgs.graph imgs.tmp
```

Determine the summation of the pixel abscissas (connected component per connected component) and store them in `param.mx`.

```
param.mx div param.area
```

Centroid abscissa is found by dividing vectors `param.mx` and `param.area` element-wise.

```
imgs.tmp clear -y; imgs.tmp cnt/add imgs.graph
param.my collect imgs.graph imgs.tmp
param.my div param.area
```

Average ordinates are computed by swapping x and y .

This overall scheme can be followed to find out second and higher-order (central) moments which are useful parameters for shape description. The following code shows how to compute the central moment μ_{20} [4]: $\mu_{20} = m_{20} - \frac{m_{10}^2}{m_{00}}$.

```

imgs.tmp clear -x; imgs.tmp ong/mult imgs.graph imgs.tmp
imgs.tmp cnt/add imgs.graph
param.mxx collect imgs.graph imgs.tmp
param.aux prod param.mx param.mx; param.aux div param.area
param.mxx sub param.aux

```

After finding the (non-central) second-order moment by squaring, only on image items, pixel x -coordinates and by collecting their sum, the central moment is then computed by working on the 1-D vectors storing moment information.

6 Conclusions

We have presented in this paper a new paradigm to deal with connected components. It relies on a high level graph-based model, called *associative nets*, that allows global manipulations of connected components. This model has been embedded in the Pacco environment leading to a very efficient tool where a large number of image analysis techniques can be easily expressed and coded. Furthermore, this scheme, although relying on high level concepts, is competitive with standard programming techniques in terms of execution time.

Source code, implementing the **Mgrp** class and related actions, is available upon request to the authors in the form of a plug-in package of the Pacco environment.

References

1. M. G. Albanesi, V. Cantoni, U. Cei, M. Ferretti and M. Mosconi, *Embedding Pyramids into Mesh Arrays*, in H. Li and Q. F. Stout (eds.) *Reconfigurable Massively Parallel Computers*, Prentice-Hall, 1990.
2. A. Biancardi, and A. Rubini, *Pacco - a new approach for an effective i.p. environment*, Proc. 12th ICPR, IEEE Press, Los Alamos, 1994, vol. 3, pp. 395-398.
3. D. Dulac, S. Mohammadi, and A. Mrigot *Implementation and evaluation of a parallel architecture using asynchronous communication*, Proc. IEEE Workshop CAMP'95, IEEE Press, Los Alamos, 1995, pp. 106-111.
4. R. C. Gonzalez, and P. Wintz *Digital Image Processing (2nd ed.)*, Addison-Wesley, Reading, 1987.
5. M. K. Hu *Visual Pattern Recognition by Moment Invariants*, IRE Trans. on Information Theory, Vol. 8, pp. 179-187.
6. *Recommendation T.6 — Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus*, International Telecommunication Union, Geneva, 1988.
7. A. Mrigot, *Associative Nets: A New Parallel Computing Model*, Tech. Report 92-02, Institut d'Electronique Fondamentale, Universit Paris Sud, 1992.
8. J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, 1994.
9. R.A. Peters, *A New Algorithm for Image Noise Reduction Using Mathematical Morphology*, IEEE Trans. on Image Processing, vol. 4, no. 5, May 1995, pp. 554-567.
10. K. Preston Jr. and M. J. B. Duff, *Modern Cellular Automata*, Plenum Press, New York, 1984.
11. J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, New York, 1982.