Implementing face recognition using a parallel image processing environment based on algorithmic skeletons

[1]Hamed Fatemi, [1]Henk Corporaal, [1]Twan Basten, [2]Pieter Jonker and [3]Richard Kleihorst

[1] Eindhoven University of Technology, PO Box 513, NL-5600 MB, Eindhoven, The Netherlands,
[2] Delft University of Technology, Lorentzweg 1, NL-2628 CJ Delft, The Netherlands,
[3] Philips Research Laboratories, Prof. Holstlaan 4, NL-5656 AA Eindhoven, The Netherlands.
h.fatemi@tue.nl

Keywords: Algorithmic skeletons, Face detection and recognition, Data parallelism, Parallel architectures.

Abstract

Image processing is widely used in many applications, including medical imaging, industrial manufacturing, and security systems. Often the size of the image is very large, the processing time has to be very small and usually real-time constraints have to be met. Therefore, during the last decades there has been an increasing interest in the development and the use of parallel algorithms in image processing.

This paper presents and evaluates a method for introducing parallelism into an image processing application. The method is based on algorithmic skeletons for low, medium and high level image processing operations. They provide an easy-to-use parallel programming interface.

To evaluate this approach, face recognition is implemented twice on a highly parallel processing platform, once via skeletons, once directly and highly optimized. It is demonstrated that the skeleton approach is extremely convenient from a programmers point of view, while the performance penalty of using skeletons is well below 10% *in our case study.*

1 Introduction

The SmartCam project [1] investigates to design a programmable smart camera for image processing applications. A smart camera combines sensors, SIMD (single instruction multiple data) processors and ILP (instruction level parallel) processors. Such a camera can be used in surveillance or inspection for detecting objects or raising an alarm when a lot of image processing should be done on-board. An SIMD processor is suitable for low-level image processing and an ILP processor can execute irregular algorithms, with intermediate and high level image processing operations such as Hough transform and labeling an object. The purpose of the project is to quantify the design of such systems via simulation and analysis in a design space exploration environment, and to develop an intuitive programming model. This model is based on algorithmic skeletons to bring parallelism into sequential code of image processing applications [8].

The goal of this paper is to define a skeleton library for image processing operations (low, intermediate and high level). By using this library, the programmer of an image processing application can easily parallelize the application and he does not have to handle the problems related to communication and synchronization. For showing the efficiency of implementing image processing applications via skeletons, we have selected face recognition as a case study. Face recognition is becoming increasingly important in image processing systems for surveillance and identification and it includes all types of image processing operations (low, intermediate and high level).

The paper is organized as follows: Section 2 puts our work in perspective with related work and explains the concept of algorithmic skeletons. The classification of skeletons for image processing applications is described in Section 3. The face recognition and detection algorithms are presented in Section 4. The implementation and evaluation of our proposed algorithms with skeletons are presented in Sections 5 and 6, followed by conclusions and future work in Section 7.

2 Related work

Skeletons are algorithmic abstractions that encapsulate different forms of parallelism, common to series of applications. The aim is to obtain environments or languages that allow easy parallel programming, in which the user does not have to handle problems related to communication, synchronization, deadlocks or non-deterministic program runs [2]. Usually, they are embedded in a sequential host language and they are used for coding and hiding the parallelism from the application user.

In [12], Serot presents a parallel image processing environment, using skeletons on top of the CAML functional language. In [10], a parallel image processing environment has been presented for low-level image operations. The skeletons have been implemented in C by using MPI as a communication library.

In this paper, we extend algorithmic skeletons to image operations of all levels (low, intermediate and high) to create a parallel image processing environment ready to use for easy implementation-development of image processing applications. Our skeleton library is embedded in the C programming language. We use this skeleton library in face recognition to show the efficiency of programming image processing applications through this library.

3 Classification and skeletonization of image operations

Image processing operations can be classified as low-level, intermediate-level and high-level [9] (Figure 1); based on this classification, it is possible to define a skeleton library for image operations.

3.1 Low-level image operations

Low-level image processing operations use the values of image pixels to modify individual pixels in an image. They can be divided into point-to-point, neighborhood-to-point and global-to-point operations [10]. Point-to-point operations depend only on the values of the corresponding pixels from the input image and the parallelization is simple. Neighborhood operations produce an image in which the output pixels depend on a group of neighboring pixels around the corresponding pixel from the input image. Operations like smoothing, sharpening, filtering, noise reduction and edge detection are highly parallelizable. Global operations depend on all the pixels of the input image, like Discrete Fourier Transform (DFT) and they are also parallelizable.

Image operations	Source	Output
Low-level	Image	Image
Intermediate-level	Image	Object/vector-data
High-level	Object/vector-data	Object/vector-data

Figure 1: Image operations

```
//skeleton for point to point operations
void PixelToPixelOp(E_IMG *in, E_IMG *out,void(*op)());

//skeleton for neighborhood to point operations
void NeighborToPixelOp(E_IMG *in, E_IMG *out,
                       E_WIN *win,void(*op)());

//skeleton for global to point operations
void GlobalToPixelOp(E_IMG *in, E_IMG *out, void(*op)());

//skeleton for image to object operations
void ImageToObject(E_IMG *in, E_OBJ *out, void(*op)());

//skeleton for object to object operations
void ObjectToObject(E_OBJ *in, E_OBJ *out, void(*op)());

//skeleton for object to point/value operations
void ObjectToPoint(E_OBJ *in, E_Point *out, void(*op)());
```

Figure 2: Skeleton library

3.2 Intermediate-level image operations

Intermediate-level image processing operations work on images and output other data structures, such as detected objects (e.g., faces) or statistics, thereby reducing the amount of information. Operations such as Hough transform (to find a line in an image), center-of-gravity calculation, labeling an object, are examples of intermediate-level image operations. They are more limited from the aspect of data parallelism when compared to low-level operations. They can be defined as image-to-object operations.

3.3 High-level image operations

High-level image processing operations work on vector data or objects in the image and return other vector data or objects. They usually have irregular access patterns and thus are difficult to run data parallel. They can be divided into object-to-object or object-to-point operations. Position estimation and object recognition theory are examples of this category.

3.4 Skeletons for image operations

It is possible to use the data-parallelism paradigm with the master-slave approach for low-level, intermediate-level and high-level image processing operations [11]. A master processor is selected for splitting and distributing the data to the slaves. The master can also process a part

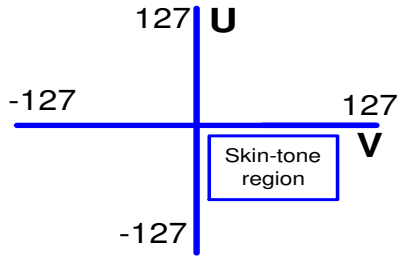

Figure 3: Skin region in UV spectrum

of the image (data). Each slave processes its received part of the image (data) and then, the master gathers and assembles the image (data) back.

Based on the above observation, we identify a number of skeletons for parallel processing of low-level, intermediate-level and high-level image processing operations. They are named according to the type of the operator. Headers of some skeletons are shown in Figure 2. Each skeleton can be executed on a set of processors. From this set of processors, a host processor is selected to split and distribute the image to the other processors. The other processors from the set, receive a part of the image and the image operation which should be applied to it. Then the computation takes place and the result is sent back to the host processor. The programmer of the image processing application should only select the skeleton from the library and gives the appropriate operation as a parameter.

4 Face detection and recognition

For recognizing a face from an image, first, it is necessary to separate it from the image, and then, it should be recognized from a data base of known faces. So, the face recognition process can be divided in two parts [3].

4.1 Face detection

For detecting faces, we have proposed in [3] an algorithm by searching for the presence of skin-tone colored pixels or groups of pixels. We have used the YUV color domain, because it separates the luminance (Y) from the true color (UV). In the RGB color space, the components represent not only color but also luminance, which varies from situation to situation (changing light causes the reliability to be decreased). By using the YUV color domain, not only the detection has become more reliable but also the skin-tone identification has become easier, because the skin tone can now be indicated in a 2−dimensional space. By measuring the UV values of human skin-tone, the skin-tone region has been identified as a rectangle in the UV spectrum (Figure 3) and every non-skin color out of the "skin box" is seen as non face (Figure 4). The face (skin) detection part, separates the skin-tone from the image and sends only the luminance and the coordinate of the skin to the recognition part. It should be mentioned that the recognition part distinguishes faces from other parts of the body such as hands and feet that have the same skin color. Furthermore, if an image is coded in the RGB color space, it is first converted to YUV.

4.2 Face recognition

The next step of the process is the recognition part. Through this process, an area of skin, detected in the previous step, is identified with respect to a face database. For this purpose, a Radial Basis Function (RBF) neural network is used [6]. The reason for using an RBF neural network is its ability to cluster similar images before classifying them [7]. An RBF neural network structure is demonstrated in Figure 5. Its architecture is similar to that of a traditional three-layer feed forward neural network. The input layer of this network is a set of n units, which accepts the elements of an n-dimensional input feature vector. (Here, the RBF neural network input is the face which is gained from the face detection part. Since it is normalized to a $64 * 72$ pixel face, it follows that $n = 4608$.) The input units are completely connected to the hidden layer with m hidden nodes. Connections between the input and the hidden layers have fixed unit weights and, consequently, it is not necessary to train them. The purpose of the hidden layer is to cluster the data and decrease its dimensionality. The RBF hidden nodes are also completely connected to the output layer. The number of the outputs depends on the number of people to be recognized (o equals the number of persons plus one; see below). The output layer provides the response to the activation pattern applied to the input layer. The change from the input space to the RBF hidden unit space is nonlinear, whereas the change from the RBF hidden unit space to the output space is linear.

For the recognition part, a skin area should be fed to the neural network input. Subsequently, the output should be calculated for each person from the database. The network node has one output node for each person from the database and the maximum value between the output nodes is considered to be the recognized person. For distinguishing a face from other parts of the body and from noise, we have preserved one of the outputs of the neural network [4].

Figure 4: Skin-tone result

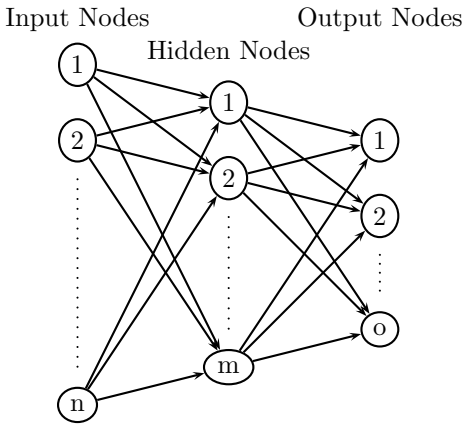

Figure 5: Architecture of RBF neural network

5 Skeletonizing

This section shows how it is possible to skeletonize image processing applications via a skeletons library. According to Section 4, face recognition can be divided into two main tasks:

- Detecting skin in the image, which can be further dived into two parts:
  - Finding the skin-tone in the image; we can map this part of the program as low-level image processing operations, because the input of this part is an image and the output is also an image.
  - Separating the skin-tones from the image as objects, and determining the coordinates of each of these skin-tones. So we map this part as intermediate-level image processing operations, because the input is an image and the output is a set of objects (faces).
- Sending each of the skin-tones (faces) to the neural network for identification, according to the faces which are in the data base. We map this part as high-level image processing operations, because the input is an object and the output is the number of the recognized person.

```
/*find skin tone*/
for (y=1; y < HEIGHT-1; y++){
        for (x=1; x < WIDTH-1; x++){
                /* convert color */
                Convertcolor(R[x][y],G[x][y],B[x][y],
                                        U[x][y],V[x][y]);
                /* find skin-tone */
                if( MIN_U<U[x][y]&&U[x][y]< MAX_U &&
                    MIN_V<V[x][y]&&V[x][y]< MAX_V)
                        out[x][y] = 1;
                else
                        out[x][y] = 0;
        }

}
/*label the image*/
for (y=1; y < HEIGHT-1; y++)
        for (x=1; x < WIDTH-1; x++)
                if(out[x][y])
                        label(label[x][y]);
/*Neural network*/
for (h=0;h< HIDDEN_NODE; h++){
        out_hidden[h]=0;
        for (i=0;i< INPUTE_NODE; i++){
                out_hidden[h] += input[i]*i2h_weight[h][i];
        }
        out_hidden[h] = ActiveFunc(out_hidden[h]);
}
for (o=0;o< OUTPUT_NODE; o++){
        out_rbf[o]= 0;
        for (h=0;h< HIDDEN_NODE; h++){
                out_rbf[o] += out_hidden[h]*h2o_weight[o][h];
        }
}
person = 0;
max = 0;
for (o=0;o< OUTPUT_NODE; o++){
        if( out_rbf[o] > max){
          max= out_rbf[o];
          person = o;
        }
}
```

Figure 6: Face recognition

Figure 6 shows the C-code of face recognition. The main parts of the program (the ones which take most time) are the parts which are inside the loops and they have the same operations for each pixel in an image or for each object (face). For being able to bring data parallelism into the program, we use skeletons as mentioned in Section 3 (Figure 2). The code can be divided into the following tasks:

- convert color: Since in our setup input is in RGB, for detecting the skin tone in the UV domain, the values of U and V should be calculated for each pixel.
- binarization: For each pixel, it should be checked whether it is within the skin-tone box or not (see Figure 3).
- labeling: For separating the faces from the image, the same label should be assigned to pixels which are nearby in the skin-tone.
- neural network: The neural network for recognizing the objects which are detected in the previous part.

The main function of the skeletonized code is shown in Figure 7 (The first three tasks are mapped onto the first three skeletons; the neural network is mapped onto the second three skeletons).

For implementation, we have used an IMAP-board [5]. In the IMAP-board (see Figure 8), the image processing is done in parallel on 256 processing elements (PEs); each PE works on one column in the image. Each PE is an 8-bit processor with 16 8-bit general purpose registers and 1 KB (kilobyte) of on-chip memory, called internal memory. At any given time, all PEs execute the same instructions (SIMD). Apart from the 1 KB of internal memory for each PE, there is 16 MB of external memory available. This memory can not be directly used by the PEs as working memory, but data can be stored in and loaded from it in order to free internal memory without losing data (swapping). This memory is generally used by the video I/O hardware to write the incoming video signal and to read the outgoing video signal. Communication between external memory and internal memory or video hardware can be done in parallel with calculations by the PEs. All elements of the card are controlled by a control processor. The control processor executes a program line by line. The program is stored in a program memory. Each line consists of a control processor instruction and a PE instruction. The control processor executes its own part of the code and sends the PE-instructions to the PEs for execution (see Figure 9).

6 Evaluation

We have implemented the skeletons library for the IMAP-board. Each implemented skeleton follows a standard template: first, the control processor reads the image or data from the external memory; then, it distributes the data between the PEs; after that, it sends the determined operations (instructions) from the skeletons to the PEs; finally, it gathers the result from the PEs and writes it in the external memory.

Figure 10 shows the execution time for each skeleton in the program. The image size that we have used is $256 * 240$ pixels and the neural network that we have used has 4608 input nodes, 15 hidden nodes and 6 (5 persons + 1 noise) output nodes.

We have also implemented a manually optimized version of face recognition (without using the skeletons) on the IMAP-board. The difference is that each skeleton reads the image (object) from the external memory, distributes it between the PEs, and again stores the results in this external memory, whereas in the optimal solution it is not always necessary to read and write data from/to the external memory. We have measured the execution time for reading, distributing and gathering an image ($256 * 240$ pixels) for

```
PixelToPixelOp(RGB, UV, &yc2ycbcr);
PixelToPixelOp(UV, skin, &Binarization);
ImageToObject(skin, obj, &labeling);

for( i= 0; i < num_object; i++){
   ObjectToObject(obj, hidden, &NeuralNet_hiddennode);
   ObjectToObject (hidden, out, &NeuralNet_outputnode);
   ObjectToValue(out, person, &Find_max);
}
```

Figure 7: Main function of skeleton code for face recognition

Figure 8: IMAP board

each skeleton and it is $0.16ms$. The average time for running each skeleton is $1.58ms$ (Figure 10). Consequently, the execution time for sending and gathering an image, takes 11% of skeleton execution time ($0.16/(1.58 - 0.16) = .11$). Note that in general, it is not necessary to send/gather the entire image to/from a skeleton (e.g. for the neural network, it's only necessary to send the skin-tone region). From these measurement, we may deduce that in general the execution time for skeletonized code is in the order of 10% worse than the execution time of an optimized program (on the IMAP-board and assuming that similar type of skeletons are used in the application). For the face recognition case study, the skeletonized code takes $8.21ms$ and the optimized code takes $7.8ms$, which is an overhead of approximately 5%.

Based on this initial experience, we expect that skeletonization can be used as a very convenient programming and implementation method which does not result in an excessive execution time overhead. It relieves the programmer from many tedious low level implementation and parallelization details.

7 Conclusions and future work

In this paper, we classified image processing operations into three categories, low, medium and high level. Based on this, we introduced and implemented an environment (library) for image processing applications based on algorithmic skeletons. By using these algorithmic skeletons, the user is completely shielded from the parallel implementation of his algorithm; he has only to provide the sequential code to process a single datum. Another advantage is that this ab-

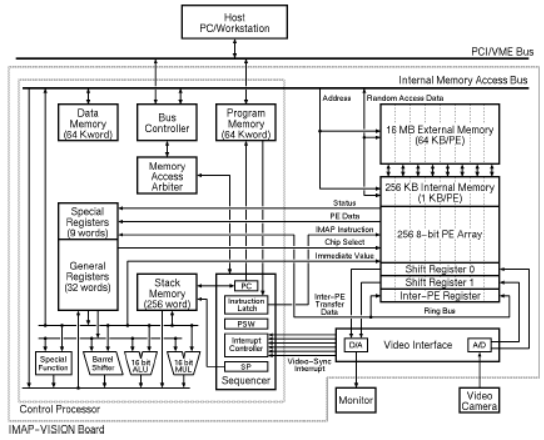

Figure 9: IMAP board architecture

Skeketon	Time(ms)
PixelToPixelOp(RGB, UV, &yc2ycbcr)	1.9
PixelToPixelOp(UV, skin,&Binarization)	1.75
ImageToObject(skin, obj, ,&labeling)	1.58
ObjectToObject(obj, hidden,&NeuralNet_hiddennode)	2.1
ObjectToObject (hiddenj, out,&NeuralNet_outputnode)	0.567

Figure 10: Execution time

straction allows the program to be executed on different processor architectures without changes to the user code. (Only the implementation of skeletons should be changed for each new architecture.)

We have also implemented face recognition via algorithmic skeletons on an IMAP-board; program implementation turned out to be straightforward and the resulting execution speed was not much worse in comparison with an optimal implementation (around 10%).

As future work, we plan to evaluate our approach by studying a larger set of case studies, implemented on a variety of parallel architectures.

References

[1] W. Caarls, P. Jonker, and H. Corporaal, "Smartcam: Devices for embedded intelligent cameras," in *PROGRESS 2002, 3rd seminar on embedded systems, Proceedings*, (Utrecht, The Netherlands), 24 October 2002.

[2] M. Cole, "Algoritmic skeletons: structured managment of parallel computations," in *Research Monographs in Parallel and Distributed Computing. MIT Press*, 1989.

[3] H. Fatemi, R. Kleihorst, H. Corporaal, and P. Jonker, "Real time face recognition on a smart camera," in *Proceedings of ACIVS 2003 (Advanced Concepts for Intelligent Vision Systems)*, (Gent, Belgium), 2003.

[4] H. Fatemi, H. E. Malek, R. Kleihorst, H. Corporaal, and P. Jonker, "Real-time face recognition on a mixed SIMD VLIW architecture," in *PROGRESS 2003, 4th seminar on embedded systems, Proceedings*, (Nieuwegein, The Netherlands), 22 October 2003.

[5] Y. Fujita, N. Yamashita, and S. Okazaki, "Imap-vision: An SIMD processor with high-speed on-chip memory and large capacity external memory," in *Proceedings of the 1996 IAPR Workshop on Machine Vision Applications*, (International Association for Pattern Recognition), 1996.

[6] J. Haddadnia, K. Faez, and P. Moallem, "Human face recognition with moment invariants based on shape information," in *Proceedings of the International Conference on Information Systems, Analysis and Synthesis*, vol. 20, (Orlando, Florida USA), International Institute of Informatics and Systemics (ISAS), 2001.

[7] Y. Hu and J. Hwang, *Handbook of neural network signal processing*. CRC Press, 2002.

[8] P. Jonker and W. Caarls, "Application driven design of embedded real-time image processing," in *Proceedings of ACIVS 2003 (Advanced Concepts for Intelligent Vision Systems)*, (Gent, Belgium), 2003.

[9] E. Komen, *Low-level Image Processing Architectures*. PhD thesis, Delft University of Technology, 1990.

[10] C. Nicolescu and P. Jonker, "Easy pipe - an easy to use parallel image processing environment based on algorithmic skeletons," in *CDROM Proceedings of Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (held in conjunction with IPDPS'2001)*, (San Francisco, U.S.A.), April 23-28, 2001.

[11] C. Nicolescu and P. Jonker, "A data and task parallel image processing environment," *Lecture Notes in Computer Science*, vol. 2131, pp. 393–408, 2001.

[12] D. Serot and J. Derutin, "Skipper: A skeleton-based programming environment for image processing applications," in *Proceeding of the Fifth International Conference on Parallel Computing Technologies*, 1999.