Multimedia Systems Manuscript Nr.
(will be inserted by hand later)

Intermediate-level vision tasks on a memory array architecture

Poras T. Balsara[1] **and Mary Jane Irwin**[2]

[1] Erik Jonsson School of Engineering & Computer Science, University of Texas at Dallas,
P.O. Box 830688 MP33, Richardson, TX 75083, USA
[2] Department of Computer Science, Pennsylvania State University, University Park, PA 16802, USA

Received January 30, 1992

Abstract. With the fast advances in the area of computer vision and robotics there is a growing need for machines that can "understand images" at very high speed. A conventional von Neumann computer is not suitable for this purpose, because it takes a tremendous amount of time to solve most typical image analysis problems. Thus, it is now imperative to study computer vision in a parallel processing framework in order to reduce the processing time. In this paper we demonstrate the applicability of a simple memory array architecture to some intermediate-level computer vision tasks. This architecture, called the *Access Constrained Memory Array Architecture* (ACMAA) has a linear array of processors which concurrently access distinct rows or columns of an array of memory modules. Because of its efficient local and global communication capabilities ACMAA is well suited to low-level as well as intermediate-level vision tasks. This paper presents algorithms for connected component labeling, determination of area, perimeter and moments of a labeled region, convex hull of a region, and Hough transform of an image. ACMAA is well suited to an efficient hardware implementation because it has a modular structure, simple interconnect and limited global control.

Key words: Memory array architecture – Computer vision tasks – Algorithms

1 Introduction

Over the last decade, the field of computer vision has evolved very rapidly, mainly because of the growing need in industrial automation, robotics and various other fields. Such vision systems need to analyze and understand images at very high speed. In reality, even the very modest image analysis tasks often require some hundreds of local neighborhood operations, thus requiring very high computation rates. This has motivated researchers to explore architectures other than the conventional von Neumann machine, which requires an excessive amount of time for sequentially processing of the input image data.

Exploiting the inherent parallelism present in various vision tasks can significantly reduce the processing time. Fortunately, with the advent of VLSI technology, hardware has become cheaper and parallelism has become increasingly affordable. This has led to several proposals and designs for parallel architectures for image processing and computer vision (Fountain et al. 1988; Ibrahim 1984; Hwang and Fu 1983; Danielson and Levialdi 1981). Construction of a parallel image processing architecture is now viable because it is possible to embed a number of processors and memory elements within a single chip in a cost-effective manner.

This paper presents the results of a study done to demonstrate the usefulness of a memory array architecture with constrained access for intermediate-level computer vision tasks. This architecture was recently proposed by Scherson and Ma (1987, 1989) and Hwang et al. (Hwang et al. 1989), who also described its use for problem solving in numerical analysis, signal processing and sorting. The architecture is based on the concept of having an orthogonally accessed memory, which was earlier proposed and used by Buehrer et al. in the EMPRESS: a dynamically configurable multiprocessor system (Buehrer et al. 1982). In this system a memory array was used as an intercommunication system to pass information among processors.

An architecture for a computer vision system should have processing capabilities for a wide range of tasks – it should be able to perform simple, neighborhood operations on the one hand and complex, global operations on the other. The memory array architecture, because of its efficient local and global communication capabilities, is well suited to a wide range of vision tasks.

In this paper we describe ACMAA algorithms for histogramming, connected component labeling, area, perimeter and moments of a region, convex hull of a region and Hough transform for extracting straight lines. A detailed description of several low-level vision algorithms

Correspondence to: Poras T. Balsara

on ACMAA is given elswhere (Balsara and Irwin 1991). In order to spare the reader from implementation details we shall assume that all algorithms in the following sections are designed for a memory array with one pixel per memory module. However, as described in Sect. 10, they can be easily modified for the case when there is more than one pixel per memory module. The algorithms described in this paper have been tested using a functional simulator for ACMAA (Kennedy 1989). This simulator was not detailed enough to provide timing and memory access information; however, it enabled us to verify the correctness of our algorithms.

In the following section we describe the organization of ACMAA and some elementary operations. Sect. 3 through 9 describe the details of a few intermediate-level vision algorithms. In Sect. 10 we discuss some features of ACMAA and these algorithms. Finally, in Sect. 11 we provide concluding remarks.

2 Access constrained memory array architecture

The organization of ACMAA of size n is depicted in Fig. 1. It consists of a linear array of n processors, ($P_i, 0 \leq i \leq n-1$) and an $n \times n$ array of memory modules, ($M_{ij}, 0 \leq i, j \leq n-1$). In order to make ACMAA applicable to a wide range of vision problems, each processor should have relatively more computation power than the simple, special-purpose processors designed for specific massively parallel computations (e.g. large $n \times n$ meshes, which are ideal for nearest neighbor computations). Typically, each ACMAA processor has a number of registers used to store local variables and is capable of performing usual Boolean (e.g., not, and, or, xor, logical-shift, etc.) and arithmetic operations (e.g., add, subtract, complement, arithmetic-shift, hardware multiply, etc.). Each memory module has a constant number of words. All the memory modules in a row i are connected to the row bus RB_i, and the modules in a column i are connected to the column bus CB_i. Each set of buses is dedicated to one processor, and there is hence no time sharing of the buses by the processors. Each processor can access a row or a column of memory. A bus controller logic takes care of switching the processors between row and column buses.

2.1 Memory access modes

As mentioned before, a processor P_i can access either memory row i or memory column i at any given time. These access modes are selected by means of a switch SW_i attached to a processor P_i.

- *Row access mode.* Each processor P_i can access any memory module j in the i^{th} row memory only, i.e., P_i can access M_{ij} for $0 \leq j \leq (n-1)$.
- *Column access mode.* Each processor P_i can access any memory module j in the i^{th} column memory only, i.e., P_i can access M_{ji} for $0 \leq j \leq (n-1)$.

Fig. 1. Organization of the ACMAA

- *Broadcast mode.* Each processor P_i can write a single data value at the same location of all the memory modules in its row or column, depending upon the state of the bus controller. This mode is useful for global initialization operations on the memory array.

The first two access modes are mutually exclusive. In our constrained model all processors operate in the same access mode and they run the same set of instructions in a lock-step fashion. Hence, it can be classified as a *Single Instruction Multiple Data* (SIMD) architecture. These access constraints remove memory access conflicts and simplify the memory control. However, this architecture could also be classified as *Multiple Instruction Multiple Data* (MIMD) with partially shared memory as mentioned by Hwang et al. (1989) if the processors are allowed to operate independently. We use this architecture in the SIMD mode, enabling us to design algorithms with simple conflict-free memory control.

In the existing literature this architecture is also referred to as *Orthogonal Multiprocessor* (Hwang et al. 1989), or *Orthogonal Memory-Access Multiprocessor* (Scherson and Ma 1987; Scherson and Ma 1989). However, we call it *Access Constrained Memory Array Architecture* (ACMAA) because it brings forth the main feature of simple control through the constraints on the orthogonal access of the memory array.

Using the above-mentioned memory access modes a processor can communicate with another processor in two steps (constant time). This is done as follows: a processor P_i communicates with another processor P_j by first putting a message in the memory module M_{ij} while operating in the *row mode*. Then processor P_j can read this message from M_{ij} by operating in the *column mode*. In a similar way every processor can send a message to every other processor simultaneously or one processor can broadcast a message to all other processors using the *broadcast mode* in two time steps or $O(1)$ time. A detailed analysis of the communication overhead introduced by the ACMAA network is presented by Scherson and Ma 1989.

2.2 Suitability of ACMAA for computer vision

ACMAA has many interesting features which make it highly suited to a wide range of tasks in computer vision. These features are mentioned below.

- *Simple structure.* The array structure and simple interconnection scheme of ACMAA are favorable characteristics for efficient hardware implementation. However, because of its simple bus-oriented interconnect structure it may not be practical to construct the ACMAA with a large number of processors because of the performance degradation due to long buses. In that case, each memory module will have more than one pixel and each processor will be responsible for a band of pixels rather than a one bit strip as described in Sect. 10.
- *Simple control.* Constrained memory access leads to simpler control than in other shared-memory architectures. A bus control logic associated with each processor enables it to access its set of row or column modules by selecting the corresponding memory modules using a memory chip select line.
- *Economical.* Most of the image processing architectures proposed so far have one processor per pixel design; hence, they are quite expensive to construct for any reasonable size gray level images. For an image of size $N \times N$ we need at most N processors and $N \times N$ size memory array to construct the ACMAA. The $N \times N$ size memory array is much cheaper and smaller to construct compared with the same size processor array. Since in any access mode only one bus accesses a memory module, it should be possible to use standard single-port memory circuits for the memory array.
- *Mapping of images.* An $N \times N$ digitized image maps directly onto the memory array; there is no restructuring or reformatting of the input data. In this respect the ACMAA has the advantage of the mesh organization over tree or pipelined architectures. Furthermore, this architecture can be easily expanded to have a stack of memory arrays instead of just one as described above, in order to store multiple images or frames in such a way that each array stores one image. This is very useful in dynamic scene analysis, and in motion analysis and detection.
- – *Efficient communications.* ACMAA provides very efficient local and global communications. Efficient local communication is useful in low-level vision tasks. Global communication through inter-processor message transfer or by a broadcast makes it well suited to intermediate-level and high-level vision tasks.

2.3 Terminology

For the algorithms described in this paper we use the notation described below:

M_{ij}	memory module in row i and column j.
$(A)^{[i,j]}$	variable A stored in memory module M_{ij}.
$(A)^{[i,*]}$	all variables A stored in i^{th} row memory modules.
$(A)^{[*,j]}$	all variables A stored in j^{th} column memory modules.
$(A)^{[i,j]}\langle k \rangle$	variable A stored in M_{ij} at location k.
$(A)^{[i,j]}\langle k, l \rangle$	variable A stored in M_{ij} at location (k, l).
A	variable A used as local variable by a processor.
Column_mode	switch to the column access mode.
Row_mode	switch to the row access mode.

3 Histogramming

In this section we provide a brief description of the ACMAA histogram algorithm of a monochrome image. A detailed description is given elsewhere (Balsara and Irwin 1991) and it is described here for the sake of completeness as it is useful for some intermediate-level vision tasks.

Let the input image $\mathbf{X}$ be of size $N \times N$ stored one pixel per memory module, and let the gray levels be quantized from 0 to $(G-1)$. For this algorithm, we assume $G \leq N$, as is the case with most images used in computer vision. The value of the histogram for a gray level g is denoted by the contents of location (bucket) H_g of the vector $\mathbf{H}$ stored one bucket per module. In the first part of our algorithm, each processor P_i computes the image histogram of row i and stores it in the vector $\mathbf{H}$ for that row. Then, in the second part all processors P_i with $i \leq G-1$ sum up the contents of all the histogram buckets H_i stored in their column memory and store the final result in the memory module M_{ii}. Details of this process are described in Algorithm 1 given below. This algorithm has a time complexity of $O(N)$ because each of its loops runs for N iterations concurrently on all the N processors. A sequential algorithm determines the histogram in $O(N^2)$ time; hence, the ACMAA algorithm is optimal for an N processor system.

4 Connected component labeling

In most computer vision problems it is desirable to break up an image into regions, each of which might correspond to different surfaces of objects or areas in a scene, etc. These regions are then labeled so that they can be identified and processed individually by some other algorithms, for example, obtaining geometric properties of regions or obtaining a connectivity graph of an image. Region labeling or connected component labeling is one of the fundamental tasks in image analysis that identifies such regions of an image. The labeling algorithm processes a two-dimensional image and assigns unique labels to different regions in the image. These regions or connected components consist of sets of pixels that are connected to each other and have the same gray value.

Algorithm 1: Image histogram **forall** *processors* P_i, $(0 \le i \le N-1)$ **do_in_parallel**

```
begin
    Row_mode;
    (H)^[i,*] = 0;              (* global initialization in each row *)
    for j = 0 to (N - 1) do        (* determine row histograms *)
        g = (X)^[i,j];
        (H)^[i,g] = (H)^[i,g] + 1;
    endfor;        Column_mode;           (* sum up the columns *)
    if (i <= G - 1) then
      for j = 0 to N - 2 do
          (H)^[N-1,i] = (H)^[N-1,i] + (H)^[j,i];
      endfor;
      (H)^[i,i] = (H)^[N-1,i];
endforall;
```

The output of this algorithm can be used by many other algorithms to further analyze the image (Bahard and Brown 1982).

Several sequential and parallel algorithms and architectures have been proposed to label regions of an input image (Lumia et al. 1983; Nassimi and Sahni 1980; Weems 1984; Ibrahim 1984; Duff and Fountain 1986; Cypher et al. 1987). Sequential algorithms based on generating a table of equivalence classes and using local neighborhood are given in (Lumia et al. 1983; Rosenfeld and Kak 1982). Although these algorithms have a time complexity of $O(N^2)$, they involve at least two scans of an image and require additional data structures and processing time to maintain and sort out a list of equivalent labels into equivalence classes. For an $N \times N$ image, Nassimi and Sahni (1980) give an $O(N)$ algorithm on a size N^2 two-dimensional mesh. This algorithm achieves good asymptotic complexity, however it uses complex data manipulation and global communication routines. Cypher et al. (1987) give an $O(N)$ mesh algorithm based on Levialdi's shrinking algorithm. To label a binary image their algorithm needs $O(\log N)$ bits of memory per pixel besides the storage required for the pixel value and its label. A straightforward implementation requires $O(N)$ additional bits of memory per pixel. Weems (1984) gives an algorithm based on local neighborhood processing on a two-dimensional mesh of content-addressable processors. This algorithm takes time proportional to the diameter of a region and for long thin regions (e.g. serpentine figures or spirals) this diameter is of the order of the area of the image $[O(N^2)]$.

The connected component labeling algorithm described in this section takes an input image array $\mathbf{X}$ of size $N \times N$ and produces an output array **label** such that $label_{ij}$ corresponds to the label of pixel x_{ij}.

Connectivity of image pixels can be described by either of these two definitions: 4-connectivity and 8-connectivity. Two pixels are 4-connected if they are adjacent vertically or horizontally (i.e., pixels at locations (i,j) and $(i,j\pm1)$ or (i,j) and $(i\pm1,j)$). They are said to be 8-connected if they are adjacent horizontally, vertically or diagonally (i.e., pixels at locations (i,j) and $(i\pm1,j\pm1)$). In this paper we shall use the definition of 4-connectivity to describe our algorithm. However, the algorithm mentioned below can be easily modified to label 8-connected pixels also.

The algorithm described in this section is based on the bottom-up, divide-and-conquer technique. We partition the input image into a number of subimages and label each subimage. Then, we merge these labeled subimages together to determine the labels of the whole image. This merging is done in a bottom-up fashion, starting from small subimages and going up to the largest subimage, i.e. the input image.

In each merge operation we merge four subimages of size $m \times m$ into one subimage of size $2m \times 2m$. Merging is done in two steps: (1) merge two subimages of size $m \times m$ horizontally across their common vertical boundary to get a subimage of size $m \times 2m$; (2) merge these subimages of size $m \times 2m$ vertically across their common horizontal boundary to get the final subimage of size $2m \times 2m$.

Merging of two subimages involves updating labels in both the subimages, so that any connected component in these subimages that extends across the common boundary is labeled by the same label. When we merge two subimages we sequentially scan the pixels along their common boundary and at each step we process a different pair of neighboring pixels. If these pixels have the same gray value then it implies that they belong to the same region. We then merge the regions corresponding to these neighboring pixels and assign a label, which is the smaller of the two pixel labels. This procedure is described in detail in Algorithm 2.

Initially, all the regions are of size 1×1 (i.e., one pixel each) and they are assigned a label, which is their index in the row major ordering of the image array. These subimages of size 1×1 are merged into larger subimages in parallel by the main loop of Algorithm 2. In this loop we first merge subimages horizontally and then vertically. Let us now examine the details of the horizontal merge operation. In this operation we merge two labeled subimages of size $m \times m$ into one subimage of size $m \times 2m$. This is done by scanning the common boundary between the two subimages pixel by pixel. The variable *vlength* keeps track of the number of boundary pixel pairs to be scanned in the present merge operation. When the i^{th} pixel pair on the boundary is considered, processor P_i is active in the row access mode. It compares the two pixels and if they have the same value it updates their labels

Algorithm 2: Region labeling **forall** *processors* P_i, $(0 \leq i \leq N-1)$ **do_in_parallel**

begin
Row_mode;
for $j = 0$ **to** $N-1$ **do** (* initialize pixel labels *)
$(label)^{[i,j]} = i * N + j$; (* $label$ = row major index *) **for** $j = 0$ **to** $\log N - 1$ **do**
$vbounds = \frac{N}{2^{j+1}}$; (* no. of boundaries *)
$hbounds = \frac{N}{2^{j+1}}$; $vlength = 2^j$; (* boundary length *)
$hlength = 2^{j+1}$; (* Merge horizontally across all vertical boundaries *)

Row_mode;
$strip = \lfloor \frac{i}{vlength} \rfloor$;
$row = i \bmod vlength$;
$vloc = vlength$; (* location of a vert. boundary *)
for $k = 0$ **to** $vbounds - 1$ **do**
for $p = 0$ **to** $vlength - 1$ **do**
if $(row = p)$ **then**
if $((X)^{[i,vloc-1]} = (X)^{[i,vloc]}) \wedge ((label)^{[i,vloc-1]} \neq (label)^{[i,vloc]})$ **then**
(* broadcast labels to both regions *)
$minl = \min\{(label)^{[i,vloc-1]}, (label)^{[i,vloc]}\}$;
$maxl = \max\{(label)^{[i,vloc-1]}, (label)^{[i,vloc]}\}$;
$(min\_label)^{[i,*]} = minl$;
$(max\_label)^{[i,*]} = maxl$;
endif;
endif;
Column_mode; (* pick up new labels from appropriate rows *)
$b = strip * vlength + p$;
$minl = (min\_label)^{[b,i]}$;
$maxl = (max\_label)^{[b,i]}$;
Row_mode; (* update labels in both regions *)
for $(b = vloc - vlength)$ **to** $(vloc + vlength - 1)$ **do**
if $((label)^{[i,b]} = maxl)$ **then**
$(label)^{[i,b]} = minl$;
endfor;
$vloc = vloc + 2 * vlength$;
endfor;

(* Merge vertically across all horizontal boundaries *)
Column_mode;
$strip = \lfloor \frac{i}{hlength} \rfloor$;
$col = i \bmod hlength$;
$hloc = vlength$; (* location of a horizontal boundary *)
for $k = 0$ **to** $hbounds - 1$ **do**
for $p = 0$ **to** $hlength - 1$ **do**
if $(col = p)$ **then**
if $((X)^{[hloc-1,i]} = (X)^{[hloc,i]}) \wedge ((label)^{[hloc-1,i]} \neq (label)^{[hloc,i]})$ **then**
(* broadcast labels to both regions *)
$minl = \min\{(label)^{[hloc-1,i]}, (label)^{[hloc,i]}\}$;
$maxl = \max\{(label)^{[hloc-1,i]}, (label)^{[hloc,i]}\}$;
$(min\_label)^{[*,i]} = minl$;
$(max\_label)^{[*,i]} = maxl$;
endif;
endif;

Row_mode; (* pick up new labels from appropriate columns *)
$b = strip * hlength + p$;
$minl = (min\_label)^{[i,b]}$;
$maxl = (max\_label)^{[i,b]}$; **Column_mode;** (* update labels in both regions *)
for $(b = hloc - vlength)$ **to** $(hloc + vlength - 1)$ **do**
if $((label)^{[b,i]} = maxl)$ **then**
$(label)^{[b,i]} = minl$;
endfor;
$hloc = hloc + 2 * vlength$;
endfor;
endfor;
endforall;

by the minimum of the two labels. Besides updating the labels of the pair it also updates all the labels of the pixels, which were connected to these two pixels. This is done as follows. Processor P_i broadcasts the smaller and the larger labels (*min_label* and *max_label*) to all the memory modules in row i (see Fig. 2a). Then in the column mode all the processors pick up these new labels from the appropriate rows of the subimage to which it belongs. That is, each processor picks up the label from the row r of the horizontal *strip* to which it belongs; e.g., in Fig. 2b processor P_c picks up labels from row b of the array (which corresponds to row r of *strip x*), whereas processor P_l picks up from row j of the array, and so on. Finally, we again switch back to the row mode and all processors scan the rows of the subimage to which they belong and update the labels. This is shown by horizontal arrows in Fig. 2c. This merging process is done in parallel for all the vertical boundaries located at the position given by the variable *vloc*, and is repeated for all such *vloc*'s in the horizontal merge operation. The total number of such boundaries is given by variable *vbounds*.

In the second part of the main loop we merge vertically along the horizontal boundaries. In this operation we merge two labeled subimages of size $m \times 2m$ into one subimage of size $2m \times 2m$. This is done in the same way as the horizontal merge operation except that now the boundaries are horizontal and they are twice the size of vertical boundaries.

The running time for Algorithm 2 can be computed by considering the number of iterations executed by the main loop and the number of steps performed in each iteration. The total number of steps is then given by,

$$T(N) = \sum_{j=0}^{\log N-1} (T_h + T_v) \tag{1}$$

where T_h and T_v are the number of steps required for horizontal and vertical merge operations.

$$T_h = \sum_{k=0}^{\frac{N}{2^{j+1}}-1} \sum_{p=0}^{2^j-1} 2^{j+1} = N * 2^j \tag{2}$$

and,

$$T_v = \sum_{k=0}^{\frac{N}{2^{j+1}}-1} \sum_{p=0}^{2^{j+1}-1} 2^{j+1} = 2N * 2^j \ . \tag{3}$$

Therefore,

$$T(N) = \sum_{j=0}^{\log N-1} 3N * 2^j = 3 * (N^2 - N) \tag{4}$$

Hence, the time complexity of the above algorithm is $O(N^2)$ for an image of size $N \times N$ on ACMAA of size N. The algorithm described above is straightforward and does not require any restructuring or preprocessing of the input data, nor does it require any complex communication routines.

Fig. 2. Embedding a pyramid on the ACMAA of size 4

5 Area of a region

The area of a region or a connected component is one of the basic descriptive properties. In a discrete image it is defined as the total number of pixels belonging to the region. A tree based, $O(\log N)$ area algorithm for the NON-VON architecture is described by Ibrahim (1984). In this section we propose two methods to compute the area of all the regions in an image. The input to our algorithms is a labeled image stored in the ACMAA memory array.

5.1 Summation pyramid approach

A pyramid is 4-ary tree constructed on an $N \times N$ array of image pixels. The leaves of this tree (at level 0) are image pixels and each internal node at level j is connected to to four nodes at level $j-1$ and one node at level $j+1$. The *apex* (root) of this pyramid is at level $\log N$. Such a pyramid can be used to perform aggregation operations on an $N \times N$ image in $O(\log N)$ time. For example, we can construct a summation pyramid by instructing each internal node to add up the values received from its four children, so that we get a sum of N^2 numbers at the apex of this pyramid.

We now show how to embed a pyramid on the ACMAA. Fig. 3 describes an embedding on ACMAA of size 4. The $N \times N$ array of memory modules forms the base of a pyramid. Since we have a two-dimensional structure to simulate a three-dimensional pyramid we shall use some of the modules to store information pertaining to more than one pyramid levels. When we merge four nodes (a 2×2 subarray) at level j into one node at level $j+1$, we store the information of this higher level parent node in the left top node of its 2×2 subarray. This is depicted in Fig. 3. In Fig. 3b the white nodes are the nodes at level 0 and the shaded nodes store information for levels 0 and 1. The darkest node in Fig. 3e corresponds to the apex at level 2 and hence it stores information of a node at all the three levels. A transfer from level j to level $j+1$

Fig. 3. Horizontal merge operation

of the pyramid is performed in two ACMAA steps as follows. In the first step, the two bottom corner nodes of each 2×2 subarray at level j send their data to the two top corner nodes using the column mode (Fig. 3a,d). Then, in the second step the right top corner node of a subarray sends the combined information to the left top corner node of its subarray, which then becomes a node at the level $j+1$ (Fig. 3b,e).

We can compute the area of a region using the pyramid embedding as follows. First, initialize a variable *sum* to 1 in all the modules (pixels) whose label corresponds to that of the region of interest. All other modules have *sum* initialized to 0. Then we construct a summation pyramid by adding up the variables *sum* in each module as we go up the pyramid so that the area of this region will be available in the memory module M_{00}, which corresponds to the apex of this pyramid. An ACMAA procedure to compute the area of a region labeled L is described in Algorithm 3.

The initialization step of Algorithm 3 takes $O(N)$ time. In order to compute the area of a component labeled L the main loop iterates $\log N$ times. In an iteration k a processor simulates $\frac{N}{2^k}$ nodes belonging to level k of the pyramid. Hence, the total time taken by the main loop is given by,

$$\sum_{k=1}^{\log N} \frac{N}{2^k} = O(N) \ . \tag{5}$$

Hence, if there are C connected components in an image, it take $O(CN)$ time steps to compute their areas using the summation pyramid approach. Area computation for a labeled region can be done on a uniprocessor in $O(N^2)$ time steps, hence the ACMAA algorithm achieves a linear speed-up using N processors.

5.2 Split-and-merge approach

In this section we describe a bottom-up, divide-and-conquer approach similar to the one described in Sect. 4 to compute the area for all regions in an $N \times N$ image on ACMAA of size N. As before, we first partition the image into subimages. Then, we compute areas in each subimage and finally we merge the subimages by adding up the areas of regions with identical labels extending across their common boundary.

Initially, all the modules have their variable *area* initialized to 1 because that is the area of one pixel stored in a module. Then in each successive iteration we merge horizontally and vertically as mentioned before. However, in the merge step we do the following. We scan the common boundary between the subimages to be merged looking for a pair of 4-connected pixels with the same label, say L. Once such a pair is found we combine the areas of this region lying in the two subimages and update the variable *area* for all modules with label L in these subimages. This new value of *area* is the sum of areas stored in the pixels of the 4-connected pair. In each merge operation we update the area of one region only once because all the modules in a subimage store the area of the component to which they belong and so we need only one 4-connected pair of the region extending across the common boundary to combine the areas from the two subimages. In Algorithm 4 we give a pseudo-code to perform the above mentioned merge operation (other parts of this algorithm are similar to that of Algorithm 2). The variable names used in this algorithm correspond to those used in Algorithm 2.

The running time for Algorithm 4 is same as that for Algorithm 2, i.e., $O(N^2)$ for an image of size $N \times N$ on ACMAA of size N. If C is the number of regions

Algorithm 3: Area computation on a pyramid (* Area computations for an $N \times N$ image on ACMAA of size N *)

```
forall processors P_i, (0 <= i <= N-1) do_in_parallel
begin
      Column_mode;
      for j = 1 to N do              (* Initialization *)
          if ((label)^[j,i] = L) then
             (area)^[j,i] = 1;
          else (area)^[j,i] = 0;
      endfor;
      for k = 1 to log N do
          Column_mode;              (* Fetch vertically (Fig.3a and d) *)
          if ((i mod 2^k = 0) ∨ (i mod 2^(k-1) = 0)) then
             for j = 0 to (N-1) step 2^k do
                (area)^[j,i] = (area)^[j,i] + (area)^[j+2^(k-1),i];
          endif;
          Row_mode;              (* Fetch horizontally (Fig.3b and e) *)
          if (i mod 2^k = 0) then
             for j = 0 to (N-1) step 2^k do
                (area)^[i,j] = (area)^[i,j] + (area)^[i,j+2^(k-1)];
          endif;
      endfor;              (* Final result is stored in module M_00 *)
endforall;
```

Algorithm 4: Merge operation for area computation ...

```
...
(* Merging horizontally across a vertical boundaries *)          for k = 0 to vbounds - 1 do
          for p = 0 to vlength - 1 do
             if (row = p) then     (* merge two regions only if they were not merged before *)
                if ((label)^[i,vloc-1] = (label)^[i,vloc]) ∧ ((update)^[i,vloc] = 0) then
                   (* combine and broadcast the two areas and their label *)
                   new_area = (area)^[i,vloc-1] + (area)^[i,vloc];
                   (n_area)^[i,*] = new_area;
                   (L)^[i,*] = (label)^[i,vloc];
                endif;
             endif;
             Column_mode;   (* pick up new area from appropriate rows *)
             b = strip * vlength + p;
             new_area = (n_area)^[b,i];
             area_label = (L)^[b,i];
             Row_mode;   (* update areas in both regions *)
             for (b = vloc - vlength) to (vloc + vlength - 1) do
                if ((label)^[i,b] = area_label) then
                   (area)^[i,b] = new_area;
                   (update)^[i,vloc] = 1;     (* set the update flag *)
                endif;
             endfor;
          endfor;
          vloc = vloc + 2 * vlength;
      endfor;
      vloc = vlength;
      for k = 0 to vbounds - 1 do
          (update)^[i,vloc] = 0;     (* reset the update flag *)
          vloc = vloc + 2 * vlength;
      endfor;

(* Similarly do it for vertical merging across all horizontal boundaries *)
...
...
```

(connected components) in an $N \times N$ image, then the summation pyramid approach gives better performance compared to the split-and-merge approach for the images in which $C \leq N$.

6 Perimeter of a region

The perimeter of a region or a connected component is defined as the total length of the region boundary, i.e., it is the total number of boundary pixels in a region. These boundary pixels include the pixels belonging to the external as well as internal (surrounding a hole) boundaries of a region. In a binary picture where background is represented by white pixels, perimeter is given by the total number of black pixels of a region adjacent to a white pixel. In this section two methods for computing perimeters of the regions in an image are proposed.

6.1 Summation pyramid approach

The summation pyramid approach used for area computation can be used to compute the perimeter of a region labeled L as follows. In the first step we mark all the boundary pixels having label L. A boundary pixel of a region is a pixel having at least one non-region pixel (i.e. a pixel whose label is not equal to L) adjacent to it in the 3×3 neighborhood around it. Then we add these marked pixels in a summation pyramid to obtain the perimeter of a region labeled L from the memory module M_{00}. Hence, for an image with C regions, it takes $O(CN)$ time on ACMAA of size N to compute all the perimeters.

6.2 Split-and-merge approach

The bottom-up, divide-and-conquer approach similar to the one described in Sects. 4 and 5.2 can also be used to compute the perimeters of all regions in an $N \times N$ image on ACMAA of size N. This is done as follows. We first partition the image into subimages, then we compute perimeters in each subimage and finally we merge the subimages by adding up the perimeters of regions with identical labels extending across the boundary between the subimages. When we merge perimeters in two subimages the perimeter pixels on the common boundary cease to be the boundary pixels because now they become internal pixels of the merged regions.

Initially all the non-background pixels are marked as boundary pixels and their variable *perimeter* is initialized to 1. Then in each successive iteration we merge horizontally and vertically as mentioned before. However, in the merge step we do the following. We scan the common boundary between the subimages to be merged looking for a pair of 4-connected boundary pixels. Once such a pair is found we merge the perimeters and then remark them appropriately as boundary or non-boundary pixels by looking at the 3×3 neighborhood around each pixel. When we merge perimeters of two regions with the same label L in the two subimages we update the variable *perimeter* of all the boundary (marked) pixels in both the subimages by the sum of two perimeters. However, when a pixel on the common boundary ceases to be a boundary pixel, we set its *perimeter* variable to 0 and subtract one from that of the other boundary pixels with the same label. After all the $\log N$ horizontal and vertical merges are performed we will have an output image in which all the marked pixels belong to boundaries of regions and the variable *perimeter* in these marked pixels correspond to the length of the boundary to which they belong.

Hence, in $O(N^2)$ time steps we can mark all the boundary pixels and compute the perimeter of all regions in an $N \times N$ image on ACMAA of size N. As mentioned before, the summation pyramid approach has a better performance compared to the split-and-merge approach for an $N \times N$ image in which the number of region $C \leq N$.

7 Moments

In computer vision moments are used as descriptors of shapes in an image. For a two-dimensional discrete function $f(x, y)$ the moment of order $(p+q)$ is defined by

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y) \ . \tag{6}$$

For a binary image, $f(x, y)$ has a value 1 inside the region and 0 elsewhere. This function reflects the shape of the object, and it has a unique set of moments (Gonzalez and Wintz 1977; Ibrahim 1984). The area of a region is given by its *zero* order $(p = q = 0)$ moments (m_{00}). m_{00} can be used to obtain the coordinates of the centroid $(\bar{x}, \bar{y})$ of a region.

$$\bar{x} = \frac{m_{10}}{m_{00}}, \quad \bar{y} = \frac{m_{01}}{m_{00}} \ . \tag{7}$$

Moments of a shape can be made size-invariant by dividing them by its area (m_{00}). They can be made position-invariant by computing *central moments* defined as follows:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y) \ . \tag{8}$$

Using these moments it is possible to derive a set of *normalized central moments* which are invariant to translation, rotation and scale change (Gonzalez and Wintz 1977).

Moments m_{pq} of a shape labeled L can be obtained by a two step ACMAA algorithm as follows: in the first step each processor computes the product $x^p y^q$ in the column mode (where $x = i$ and $y = j$ for processor P_i accessing the j^{th} module in its column memory). This can be done in $O(N)$ time steps for an image of size $N \times N$ on ACMAA of size N. In the second step we first multiply these products with their respective function values $f(x, y)$ (where, $f(x, y) = \{1 | label(x, y) = L\}$) and then add these new products over the whole image using the summation pyramid approach described earlier. Hence, the complete algorithm takes $O(N)$ time steps to compute $(p+q)$ order moments of a shape (region) labeled L. Note that for computing the moments m_{pq} for all the regions in an image the first step needs to be executed only once because the product $x^p y^q$ does not depend on whether the point (x, y) lies inside a region or elsewhere.

8 Convex hull of a region

In this section we describe an algorithm to determine vertices of the convex hull of a region consisting of pixels

Fig. 4. Determining extreme points of the convex hull of a region

with the same label. Convex hull of a region is the smallest convex polygon containing all the pixels belonging to the region. We assume that we have a labeled image stored in the memory array, one pixel per module. Given a set S of pixels with label L, we define $Hull(S)$ as a set of pixels of S that are vertices of the convex hull. Sides of this convex polygon can be easily determined from this set of vertices (also refered to as *extreme points of* S). Miller and Stout (1985) show that many queries concerning S can be reduced to questions concerning the extreme points of S. Fig. 4a shows convex hull of a region consisting of pixels with the same label (shaded).

In order to determine if a pixel of region S belongs to $Hull(S)$, we make use of the following property of an extreme point of a convex hull: let, $x \in S$ be a point and β be the smallest angle that encloses S subtended at x, then, $x \in Hull(S)$ iff $\beta < 180°$. For example, in Fig. 4a point $p \in Hull(S)$ because $\beta < 180°$ whereas, points q and r in Fig. 4b and c do not belong to $Hull(S)$ because $\beta \geq 180°$.

In our algorithm each processor first determines the extreme points in its column. These points are likely candidates for extreme points of the convex hull. On an $N \times N$ array of pixels, S may have $O(N^2)$ points, but since S has at most two extreme points in any column, it has $O(N)$ extreme points. Each processor broadcasts its extreme points to all other processors which then determine if their extreme points belong to $Hull(S)$ using the property mentioned above. Algorithm 5 describes a procedure to determine $Hull(S)$ for a region S, which is made up of pixels with label L.

In Algorithm 5 each processor determines its topmost and bottomost points in $O(N)$ time steps by scanning its column memory. Then each processor broadcast its *top* and *bottom* values to all its row modules using the broadcast mode. This is done in $O(1)$ time. Finally, each processor determines whether the extreme points of its column belong to $Hull(S)$ by scanning the points on its right and left. This takes $O(N)$ time steps. Therefore, extreme points of the convex hull of a region can be identified in $O(N)$ time steps.

9 Hough transform

Hough transform is a technique used to extract straight line segments in an image. Line segments are frequently used as key features in constructing image models. This method involves transforming each image point into a sinusoidal curve in a *parameter space* defined by Duda and Hart 1972.

$$\rho = x \cos\theta + y \sin\theta \ . \tag{9}$$

In this parameterization (x, y) is a point on the line, ρ is its normal distance from the origin and θ is the angular position of the normal. This parameter space is also referred to $\rho - \theta$ *plane* or *Hough space*.

In order to keep the parameters for a line unique, θ is restricted to the interval $[0, \pi]$ (Duda and Hart 1972). This restriction gives us a unique point in the parameter plane for every line in the image plane. Hence, for an $N \times N$ image and $0 \leq \theta < \pi$ we get, $-\sqrt{2}N \leq \rho \leq \sqrt{2}N$. Let the parameter plane be quantized into a quadruled grid, and the quantized region be treated as a two-dimensional accumulator array. For each point (x_i, y_i) in the image plane, the corresponding curve in the parameter plane is entered in this array by incrementing the count in all cells along the sinusoid. The peak count k in a cell (θ_i, ρ_j) implies that there are k collinear points along the line whose parameters are (θ_i, ρ_j).

Let $\Delta\theta$ and $\Delta\rho$ be the resolutions for θ and ρ, then the size of the Hough accumulator array is $R \times \Theta$, where

$$\Theta = \frac{\pi}{\Delta\theta} \quad \text{and} \quad R = \frac{2\sqrt{2}N + 1}{\Delta\rho} \tag{10}$$

ρ can be quantized in the same manner as x and y, whereas $\Delta\theta$ is chosen as a compromise among speed, space and accuracy.

The line extraction algorithm based on Hough transform can be divided into two parts: computation of the count values in the Hough array, and determination of the parameter cells whose counts exceed a certain threshold. Computation of the Hough array is done sequentially for all values of θ. For each value of θ a processor P_i computes the contributions of all the pixels stored in its column memory and stores the results in a vector r_θ which is of size R. These column vectors are then added up and the resulting vector is stored in the memory column corresponding to the value of θ. Algorithm 6 describes the details of the Hough transform procedure for extracting line segments.

The second part of the line extraction algorithm is responsible for reporting the detected lines in the input image (X) by post-processing the Hough array. This can be achieved by a number of ways, for example, it can be done by reporting the contents of the Hough cells

Algorithm 5: Extreme points of convex hull of a region **forall** *processors* P_i, $(0 \leq i \leq N-1)$ **do_in_parallel**

begin
  Column_mode;
  $top = \infty$; (* Initializations *)
  $bottom = -\infty$; (* Determine extreme points for each column of region with label L *) **for** $j = 0$ **to** $N-1$ **do**
    if $(top = \infty) \wedge ((label)^{[j,i]} = L)$ **then**
      $top = j$;
    if $((label)^{[j,i]} = L)$ **then**
      $bottom = j$;
    $(Hull)^{[j,i]} = .false.$; (* Initialize $Hull(S)$ *)
  endfor;
  Row_mode; (* Broadcast *top* and *bottom* to all columns *)
  $(Top)^{[i,*]} = top$;
  $(Bottom)^{[i,*]} = bottom$; (* Determine if the column extreme points belong to $Hull(S)$ *) **Column_mode;**
  (* Check the bottom points with $bottom \neq -\infty$ *)
  $\alpha = \infty$;
  for $j = i+1$ **to** $N-1$ **do** (* points on the right *)
    $\delta x = j - i$;
    $\delta y = bottom - (Bottom)^{[j,i]}$;
    $\theta = \tan^{-1}(\frac{\delta y}{\delta x})$;
    $\alpha = \min\{\alpha, \theta\}$; (* *bottom* point with most negative (smallest) α *)
  endfor; $\gamma = -\infty$;
  for $j = i-1$ **to** 0 **step** -1 **do** (* points on the left *)
    $\delta x = j - i$;
    $\delta y = bottom - (Bottom)^{[j,i]}$;
    $\theta = \tan^{-1}(\frac{\delta y}{\delta x})$;
    $\gamma = \max\{\gamma, \theta\}$; (* *bottom* point with most positive (largest) γ *)
  endfor; $\beta = 180^\circ + \gamma - \alpha$;
  if $(\beta < 180^\circ)$ **then**
    $(Hull)^{[bottom,i]} = .true.$; (* Check the top points with $top \neq \infty$ *)
  $\alpha = -\infty$;
  for $j = i+1$ **to** $N-1$ **do** (* points on the right *)
    $\delta x = j - i$;
    $\delta y = top - (Top)^{[j,i]}$;
    $\theta = \tan^{-1}(\frac{\delta y}{\delta x})$;
    $\alpha = \max\{\alpha, \theta\}$; (* *top* point with most positive (largest) α *)
  endfor;
  $\gamma = \infty$;
  for $j = i-1$ **to** 0 **step** -1 **do** (* points on the left *)
    $\delta x = j - i$;
    $\delta y = top - (Top)^{[j,i]}$;
    $\theta = \tan^{-1}(\frac{\delta y}{\delta x})$;
    $\gamma = \min\{\gamma, \theta\}$; (* *top* point with most negative (smallest) γ *)

  endfor; $\beta = 180^\circ - \gamma + \alpha$;
  if $(\beta < 180^\circ)$ **then**
    $(Hull)^{[bottom,i]} = .true.$; **endforall;**

that are greater than a certain threshold and their corresponding (ρ, θ) values; or lines can be shown graphically in an output image. For the first method each processor can scan its row and report the values that exceed a threshold value. Procedure for the second approach is slightly elaborate and it is described below in Algorithm 6. The procedure is as follows: For each value of θ we first broadcast the Hough array column to all the memory array columns. Then, each processor marks its pixels "1" if they contribute to the Hough accumulator cell whose count exceeds a certain threshold. Once a pixel is marked "1" it is never made "0". The output image (L) thus created has the extracted straight lines of the input image. Sometimes this simple thresholding of the Hough array may not be enough to extract lines because there can be a lot of clustered cells in the Hough array exceeding the threshold. In that case we may need to further process the array using a local maxima computation and then report the line segments. Local maxima computations on the ACMAA can be done using local neighborhood operations similar to those described in Balsara and Irwin (1991).

The first section of Algorithm 6 computes the columns of the Hough array for each value of θ sequentially. Hence, we perform Θ many iterations. In an iteration t every processor computes the contributions of pixels stored in its column memory and stores the results in the vector r_θ. This takes $O(N)$ time. Finally, addition of all these vectors and storing the result in the column corresponding to θ_t takes $O(N)$ time. Hence, the Hough array is created in $O(\Theta N)$ time. On the similar lines extraction of straight lines in the second section also takes $O(\Theta N)$ time. Hence, the total time for Algorithm 6 is (ΘN).

A sequential algorithm takes $O(\Theta N^2)$ time steps to compute the Hough transform of an $N \times N$ image. Hence, our algorithm achieves a linear speed-up and that, it is optimal for N processors. The ACMAA algorithm achieves the same running time as the mesh algorithm described in Silberberg (1985). However, we use only N processors compared to N^2 processors in the mesh.

10 Discussion

Algorithms described in this paper are simple and easy to understand. There are no complex pointer manipulations or data routing schemes. All operations are performed without any reformatting or serialization of the input image. This leads to a straightforward implementation with limited global control. Also, it results in a time efficient implementation for the tasks in which more than one operation are performed on an input image because no rearranging or input/output of the intermediate results is needed. All of the algorithms work on both, binary as well as gray scale images. However, the computation time for the latter will be more than that for the former because of the increase in operand size.

Table 1 summarizes all the algorithms (for $N \times N$ images) described in this paper and compares the ACMAA time complexities with those of the other architectures. A comparison of these architectures with ACMAA is described in Table 2.

All the above algorithms perform their operations on images of size $N \times N$ using ACMAA of size N. They can be easily generalized for large size images ($N > n$, where n is the size of ACMAA). However, this requires memory modules of larger size, and will slow down the algorithms by a constant factor. This modification is done by storing an image block of size $\lceil \frac{N}{n} \rceil \times \lceil \frac{N}{n} \rceil$ in each memory module, so that each processor is responsible for a strip of the image rather than a one pixel wide row or column. Similar blocking scheme is employed for storing the histogram if $G > n$ and the Hough accumulator array. Hence, the algorithms described above can be used for images whose size is not limited by the number of processors in the ACMAA. This scheme is useful since it is not practical to construct ACMAA with a large number of processors because of performance degradation due to long buses. A modified pseudo-code for the case when $N > n$ and $G > n$ in the histogram algorithm described in Sect. 3 is given below. In this algorithm, the input image and the histogram array are stored using a block storage scheme. In this scheme we store a subimage of size $k \times k$ (where $k = \lceil N/n \rceil$) in each memory module. Hence, the element x_{ij} of an image X is stored in the memory module (subimage), $M_{\lfloor i/k \rfloor \lfloor j/k \rfloor}$ at the block location $\langle i \bmod k, j \bmod k \rangle$. In the pseudo-code below, the notation $(X)^{[i,j]}\langle r, s \rangle$ corresponds to the element of an array X stored in the memory module M_{ij} at the location (r, s).

Table 2. Comparison of different architectures

	2d Mesh	Binary Tree	Pyramid	ACMAA
# Processors	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$
Memory size	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Diameter	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$

ACMAA's efficient local and global communication schemes make it suitable for neighborhood processing (Balsara and Irwin 1991) as well as for global aggregation of values at a number of image points. Hence, it is applicable for a wide range of computer vision tasks. Also, it is economical compared to the other architectures proposed for image processing and computer vision, viz., mesh, tree, pyramid. It needs fewer processors than these architectures. Table 2 gives a comparison of ACMAA with these architectures (based on one pixel per processor or memory module scheme for an $n \times n$ image).

Currently, we are in the process of designing and implementing a 4-processor prototype of ACMAA using off-the-shelf components. Since ACMAA is an SIMD architecture it is controlled by a host processor which is responsible for the global control of the architecture and interaction with the outside world. This host processor ensures that all processors execute the same set of instructions in a lock-step fashion, thus guaranteeing SIMD mode of operation. Image input/output to/from the memory array can be done by scanning an image frame buffer row by row, left to right and using the row buses to access the memory array. The control and addressing information in the input/output mode is generted by the host. For high-speed operations one can construct ACMAA with more than one memory array so that when the processors are operating on one array, the host can perform input/output on the other.

For an ACMAA based on 64 Motorola 68008 processors with a 8MHz clock our preliminary calculations predict a running time of 12.028 ms for the histogram algorithm (Sect. 3) running on a 512×512 size image memory with an access time of 200 ns. The corresponding time for a sequential algorithm running on the same processor was 721.248 ms. Note that these approximate timings are obtained by manually determining the number and type of instructions needed to execute the algorithm. Actual timings will depend upon the components used and their layout in the system, especially that of the buses. Memory circuits used in the architecture will play an important role in determining the performance of ACMAA because too many accesses to the memory array may hinder its use for high-speed computations. Massively parallel architectures, e.g., $O(N^2)$ mesh, are normally designed to have very few memory accesses. However, they are inefficient for global communications,

Algorithm 6: Hough transform (* Array X = edge image, array L = line image. Hough array of size $R \times \Theta$ is stored in block storage fashion; i.e., more than one element per memory module.
$R = 2\sqrt{2}N + 1$, Θ =total no. of θs, $\Delta\theta$ =resolution for θ *) **forall** *processors* P_i, $(0 \leq i \leq N-1)$ **do_in_parallel**

begin
  Column_mode; (* Initializations *)
  $r = \lceil \frac{R}{N} \rceil$;
  $s = \lceil \frac{\Theta}{N} \rceil$;
  for $j = 0$ **to** $(N-1)$ **do**
    $(L)^{[j,i]} = 0$;
    for $k = 0$ **to** $(r-1)$ **do**
      for $l = 0$ **to** $(s-1)$ **do**
        $(H)^{[j,i]}\langle k, l\rangle = 0$;
        $(r_\theta)^{[j,i]}\langle k\rangle = 0$;
      endfor;
  endfor; (* $\forall$ pixels compute ρ values for every angle θ *) **for** $t = 0$ **to** $(\Theta - 1)$ **do**
    $\theta = t.\Delta\theta$;
    for $j = 0$ **to** $(N-1)$ **do** (* $\forall$ pixels in column i *)
      if $((X)^{[j,i]} = edge)$ **then**
        $\rho = i\cos\theta + j\sin\theta$;
        $p = \lfloor(\rho + \sqrt{2}N)\rfloor$;
        $(r_\theta)^{[\lfloor\frac{p}{r}\rfloor, i]}\langle p \bmod r\rangle = (r_\theta)^{[\lfloor\frac{p}{r}\rfloor, i]}\langle p \bmod r\rangle + 1$;
      endif;
    endfor; (* Add similar ρ's and store them in the column corresponding to θ_t *) **Row_mode;**
    $col = \lfloor\frac{\theta}{s}\rfloor$;
    $l = \theta \bmod s$;
    for $k = 0$ **to** $(r-1)$ **do**
      for $j = 0$ **to** $(N-1)$ **do**
        $(H)^{[i,col]}\langle k, l\rangle = (H)^{[i,col]}\langle k, l\rangle + (r_\theta)^{[i,j]}\langle k\rangle$;
        $(r_\theta)^{[i,j]}\langle k\rangle = 0$; (* reset for next θ *)
    endfor;
    Column_mode;
  endfor; (* Extract lines by enhancing those pixels that contribute to
    accumulator cells with counts exceeding the threshold. *)
  for $t = 0$ **to** $(\Theta - 1)$ **do**
    $\theta = t.\Delta\theta$;
    $col = \lfloor\frac{\theta}{s}\rfloor$;
    $l = \theta \bmod s$;
    Row_mode; (* broadcast ρ's for this θ *)
    for $k = 0$ **to** $(r-1)$ **do**
      $(r_\theta)^{[i,*]}\langle k\rangle = (H)^{[i,col]}\langle k, l\rangle$;
    Column_mode;
    for $j = 0$ **to** $(N-1)$ **do**
      $\rho = i\cos\theta + j\sin\theta$;
      $p = \lfloor(\rho + \sqrt{2}N)\rfloor$;
      if $((r_\theta)^{[\lfloor\frac{p}{r}\rfloor, i]}\langle p \bmod r\rangle \geq THRESH)$ **then**
        $(L)^{[j,i]} = 1$;
    endfor;
  endfor;
endforall;

and since the number of processors is large, an individual processor has limited local memory. These reasons may cause deterioration in the performance of such architectures for problems requiring large memories and efficient communications. ACMAA, on the other hand, is efficient for local as well as global communications but involves accesses to the memory array. Hence, the choice of approach was mainly governed by the need to use an architecture which will be suitable for a wide range of vision tasks.

11 Conclusion

Parallel algorithms for intermediate-level computer vision for the Access Constrained Memory Array Architecture (ACMAA) have been presented in this paper. The choice of algorithms demonstrates the applicability of ACMAA to several computer vision and image analysis tasks that involve local as well as global processing. This was done by exploiting the row and column bus structure of the memory array, which led to efficient local and global communications among the processors. Most of the ACMAA algorithms achieve a linear speed

Table 1. Comparison of algorithms on various architectures

Algorithm	Running time			
	Sequential machine	2d Mesh	Tree NON-VON[1]	ACMAA
Histogramming	$O(N^2)$	$O(N)$	$O(G + \log N)$	$O(N)$
Connected Component Labeling	$O(N^2)$	$O(N)$	$O(N)^2$	$O(N^2)$
Area of a Region	$O(N^2)$	$O(N)$	$O(\log N)$	$O(N)$
Perimeter of a Region	$O(N^2)$	$O(N)$	$O(\log N)$	$O(N)$
Moments of a Region	$O(N^2)$	$O(N)$	$O(\log N)$	$O(N)$
Convex Hull of a Region	$O(N^2)^3$	$O(N)$	–	$O(N)$
Hough Transform	$O(\Theta N^2)$	$O(\Theta N)$	–	$O(\Theta N)$

[1] Computed from Ibrahim 1987
[2] This is the average case running time based on the fact that the average number of black rectangles in the binary image tree is $O(N)$ (Ibrahim 1987)
[3] The input is given as a 2-d array of points and not as a list of (x, y) coordinates of points

Algorithm 7: Image histogram for $N > n$, and $G > n$ (* Initially we store the histogram bucket H_m in the column module, $M_{i,\lfloor m/k \rfloor}$ of all the n rows *)

forall *processors* P_i, $(0 \le i \le n-1)$ **do_in_parallel**
begin
    Row_mode;
    $w = \lceil \frac{G}{n} \rceil$;
    for $p = 0$ **to** $(w-1)$ **do**
        $(H)^{[i,*]}\langle p \rangle = 0$; (* global initialization in each row *)

    for $j = 0$ **to** $(n-1)$ **do** (* determine histogram for row of sub-images *)
        for $p = 0$ **to** $(\lceil \frac{N}{n} \rceil - 1)$ **do**
            for $q = 0$ **to** $(\lceil \frac{N}{n} \rceil - 1)$ **do**
                $g = (X)^{[i,j]}\langle p, q \rangle$;
                $(H)^{[i,\lfloor g/w \rfloor]}\langle g \bmod w \rangle = (H)^{[i,\lfloor g/w \rfloor]}\langle g \bmod w \rangle + 1$;
            endfor;
        endfor;
    endfor;

    Column_mode; (* sum up the columns having histogram array, H *)
    if $(i \le \lfloor \frac{G-1}{n} \rfloor)$ **then**
        for $j = 0$ **to** $n-1$ **do**
            for $p = 0$ **to** $(w-1)$ **do**
                if $(j \ne i)$ **then**
                    $(H)^{[i,i]}\langle p \rangle = (H)^{[i,i]}\langle p \rangle + (H)^{[j,i]}\langle p \rangle$;
            endfor;
        endfor;
    endif;
endforall;

up over sequential algorithms and they are optimal for an N processor system.

References

1. Ballard D, Brown C (1982) Computer vision. Prentice-Hall Inc., Englewood Cliffs, New Jersey
2. Buehrer R (1982) The ETH-Multiprocessor EMPRESS: A dynamically configurable MIMD system. IEEE Trans Comput C-31(11):1035–1044
3. Balsara PT, Irwin MJ (1991) Image processing on a memory array architecture. VLSI Signal Proc (2):313–324
4. Cypher R, Sanz JLC, Snyder L (1987) Practical algorithms for image component labeling on SIMD mesh connected computers. In: Proceedings of the International Conference on Parallel Processing, p 772–779
5. Duff MJB, Fountain TJ (ed) (1986) Cellular logic image processing. Academic Press, London, England
6. Duda R, Hart P (1972) Use of the hough transform to detect lines and curves in pictures. Commun ACM 15(1):11–15
7. Danielsson P, Levialdi S (1981) Computer architectures for pictorial information systems. IEEE Computer, pp 53–67
8. Fountain TJ, Matthews KN, Duff MJB (1988) The CLIP7A Image processor. IEEE Trans Pattern Anal Mach Intell 10(3):310–318
9. Gonzalez R, Wintz P (1977) Digital image processing. Addison-Wesley, Reading
10. Hwang K, Fu K (1983) Integrated computer architectures for image processing and database management. IEEE Computer, pp 51–60
11. Hwang K, Tseng PS, Kim D (1989) An orthogonal multiprocessor for large-grain scientific computations. IEEE Trans Comput 38(1)

12. Ibrahim HAH (1984) Image understanding algorithms on fine-grained tree-structured SIMD machines. PhD thesis, Computer Science Dept., Columbia University

13. Ibrahim HAH, Kender J, Shaw DE (1987) Low level image analysis tasks on fine-grained tree-structured SIMD machines. Parallel Distrib Comput 4:546–574

14. Kennedy J (1989) Simulator for a memory array architecture, M.S. Paper, Dept. of Computer Science, Penn State University, University Park, PA 16802

15. Lumia R, Shapiro L, Zumiga O (1983) A new connected component algorithm for virtual memory computers. Comput Grap Image Proc 22:287–300

16. Miller R, Stout QF (1985) Geometric algorithms for digitized pictures on a mesh-connected computer. IEEE Trans Pattern Anal Mach Intell, PAMI-7(2):216–228

17. Nassimi D, Sahni S (1980) Finding connected components and connected ones on a mesh connected parallel computer. SIAM J Comput 9(4):744–757

18. Rosenfeld A, Kak AC (1982) Digital picture processing. Academic Press, New York

19. Silberberg T (1985) The Hough transform on the geometric arithmetic parallel processor. In: Proceedings of the IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management, pp 387–393

20. Scherson I, Ma Y (1987) Vector computations on an orthogonal memory access multiprocessing system. In: 8th IEEE Symposium on Computer Arithmetic, pp 28–37

21. Scherson I, Ma Y (1989) Analysis and applications of orthogonal access multi-processor. J Parallel Distrib Comput 7(2):232–255

22. Weems C (1984) Image processing on a content addressable array parallel processor. Technical Report COINS Tech. Report 84-14, University of Massachusetts, Amherst

SON PHAM is currently a professor of Computer Science, College of Engineering and Computer Science at the California State University at Northridge. His research interests include Computer Graphics and Software Engineering. He has published a dozen refereed academic/technical papers in Computer Science. He is a member of ACM. Pham received his BA in Mathematics from the University of Saigon, VietNam in 1973; MA in Mathematics from University of Louisville, Kentucky in 1975; Ph.D. in Statistics from the University of Cincinnati, Ohio in 1978; and PD in Computer Science from the University of California, Berkeley in 1980.

MEL SLATER joined the Department of Computer Science, Queen Mary and Westfield College (University of London) in 1981. Since 1982 he has been involved in teaching and research in computer graphics. He produced an early implementation of the GKS graphics standard, and was involved in the ISO graphics group, with responsibility for the PASCAL language binding to GKS. He co-authored an undergraduate textbook on computer graphics, which was published in 1987. He is a principal investigator in several funded research projects, in particular the European Community ESPRIT funded SPIRIT Workstation project, and a project on Virtual Reality with the London Parallel Applications Centre. He was visiting professor in the Computer Science Division of Electrical Engineering and Computer Sciences, University of California at Berkeley in the spring semesters 1991 and 1992.

W. Kenneth Stewart is an assistant scientist at the Deep Submergence Laboratory of the Woods Hole Oceanographic Institution. His research interests include underwater robotics, autonomous vehicles and smart ROVs, multisensor modeling, real-time acoustic and optical imaging, and precision underwater surveying. Stewart has been going to sea on oceanographic research vessels for 19 years, has developed acoustic sensors and remotely-operated vehicles for 6000-m depths, and has made several deep dives in manned submersibles, including a 4000-m excursion to the *USS Titanic* in 1986. He is a member of the Marine Technology Society, Oceanography Society, IEEE Computer Society, ACM SIGGRAPH, and NCGA. Stewart received a PhD in oceanographic engineering from the Massachusetts Institute of Technology and Woods Hole Oceanographic Institution Joint Program in 1988, a BS in ocean engineering from Florida Atlantic University in 1982, and an AAS in marine technology from Cape Fear Technical Institute in 1972.

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.