Efficient Image Processing Algorithms on the Scan Line Array Processor[1]

David Helman
Department of Electrical Engineering
University of Maryland
College Park, MD 20742

Joseph JáJá
Institute for Advanced Computer Studies
Department of Electrical Engineering
Institute for Systems Research
University of Maryland
College Park, MD 20742

Abstract

We develop efficient algorithms for low and intermediate level image processing on the scan line array processor, a SIMD machine consisting of a linear array of cells that processes images in a scan line fashion. For low level processing, we present algorithms for block DFT, block DCT, convolution, template matching, shrinking, and expanding which run in real-time. By real-time, we mean that, if the required processing is based on neighborhoods of size $m \times m$, then the output lines are generated at a rate of $O(m)$ operations per line and a latency of $O(m)$ scan lines, which is the best that can be achieved on this model. We also develop an algorithm for median filtering which runs in almost real-time at a cost of $O(m \log m)$ time per scan line and a latency of $\lfloor \frac{m}{2} \rfloor$ scan lines. For intermediate level processing, we present optimal algorithms for translation, histogram computation, scaling, and rotation. We also develop efficient algorithms for labelling the connected components and determining the convex hulls of multiple figures which run in $O(n \log n)$ and $O(n \log^2 n)$ time, respectively. The latter algorithms are significantly simpler and easier to implement than those already reported in the literature for linear arrays.

[1]Supported in part by the National Science Foundation, Grant No. CCR-99103135 and by the National Science Foundation Engineering Research Center Program NSF DCD-8803012.

1 Introduction

1.1 Motivation

Real-time image processing and understanding has long been regarded as a particularly demanding problem for computer implementation, both because of the computational complexity and because of the large I/O bandwidth required by most of the tasks involved. Consider, as an example, the I/O bandwidth required to perform real-time HDTV simulation. Such a task typically involves the handling of 1K by 1K frames at the rate of 60 frames per second and results in a bandwidth requirement of approximately 500 Mbytes per second for a progressively scanned image. Not surprisingly, such problems generally lie well beyond the capacity of existing sequential processors. Consequently, a great deal of effort has been devoted to developing parallel architectures and algorithms for real-time image processing.

The simplest category of the proposed architectures is the two-dimensional array, or mesh. Examples of this class include the MPP [1], the CLIPP series [2], the MasPar [3], the DAP [4], and the GAPP [5]. The general intent behind these SIMD (Single Instruction Stream, Single Data Stream) machines is that the dimensions of the mesh should match those of the input image, and that the pixels should be assigned to processors so as to maintain the spatial relationships of the image. As a consequence, these machines can perform the local window operations typical of low level image processing with extreme efficiency. Of course, there is a considerable cost usually associated with such a large number of processors. Furthermore, while nearest-neighbor links might make local inter-processor communication quite fast, communication between two processors on either end of an $n \times n$ array will require $\Omega(n)$ time.

To reduce the cost of global communication while retaining the advantages of the mesh, a second class of architectures has been proposed. This category improves on the mesh by linking it to a sequence of progressively smaller meshes, each mesh having dimensions that are one half those of its predecessor. As a result of this pyramidal structure, the global communication costs on an $n \times n$ array can be reduced from $\Theta(n)$ to $\Theta(\log n)$ steps. Thus, this pyramidal architecture is more appropriate for intermediate level tasks which require the global exchange of information. Examples of this pyramidal machine include the PAPIA [6] and the GAM [7] systems. Of course, when compared to the mesh, there is an increased cost associated with the pyramid due to its increased complexity.

In general, high level image processing requires the simultaneous execution of several independent tasks, which is more appropriate for a MIMD (Multiple Instruction Stream, Multiple Data Stream) machine. But the disadvantage with using a purely MIMD machine is that, while they can simulate SIMD operation, they do so only with a considerable overhead for control and synchronization. Hence, they can not be expected to perform the data parallel computations typical of low level image processing with the same efficiency displayed by truly SIMD machines. Recognizing the relative advantages of SIMD and MIMD operations, a number of machines have been proposed which generally consist of an elaborate combination of reconfigurable SIMD and MIMD modules. The basic idea behind creating these hybrids is to enable the programmer to utilize the most efficient architecture for whatever particular problem is presented. Examples of this class of architectures include the Image Understanding Architecture (IUA) [8], the Associative String Processor (ASP) [9], and NETRA [10]. While this strategy for architecture design probably offers the best hope for achieving optimal performance across the spectrum of processing tasks, it will also be the most expensive.

Fortunately, not all applications require all levels of image processing. For those situations which are limited to low and intermediate level tasks, either of the SIMD architectures already mentioned might well be sufficient. However, there is one particularly simple example that avoids most of the cost constraints mentioned so far and that can still achieve optimal performance for low and intermediate level operations. This architecture has been proposed and in some cases implemented under a variety of names, including the Scan Line Array Processor (SLAP) [11] [12], the Princeton Engine [13], and the Sarnoff Engine [14]. From here on, this architecture will be referred to as the SLAP. The basic topology of this SIMD machine is a linear array of processors, in which the number of processors corresponds to the number of pixels in each row of the image. Incoming data is loaded line by line into the processor array, with a distinct processor receiving each image column. This of course results in a very high I/O bandwidth. The memory in this architecture is entirely distributed among the individual processors, and each processor can communicate directly only with its two immediate neighbors, whenever they exist. The most obvious disadvantage with this interconnection scheme is the very small communication bandwidth, which would seem to make global communication very inefficient. However, there are a number of compensations when compared with the alternatives. Anything else would involve either a more elaborate interconnection network or a shared memory, with an accompanying increase in cost and complexity. Further, scaling the SLAP simply requires adding more processors to the end of the existing array, and unlike bus based systems there are no obvious technical limitations on how many such processors can be linked together without degrading the performance. Hence, it seems reasonable to suspect that this linear array architecture might offer one of the lowest cost options for achieving optimal low and intermediate level image processing. However, for this to be true, it is necessary to argue not just that the technology is relatively cheap, but that efficient algorithms can be written to run on this architecture.

Surprisingly, the design of algorithms which utilize the linear array architecture has received only modest attention to date in the literature. Specifically, algorithms have been proposed for sorting [15], matrix multiplication [16], and the Hough transform [17]. Algorithms have also been proposed to solve a number of other graph [18] and geometric problems [19], but only by assuming that the image data is already partitioned among the processors according to the shuffled row-major distribution. The advantage of this shuffled row-major distribution is that it significantly reduces the cost of global communication between the different contiguous regions of the image. The disadvantage is that it is rather time consuming to implement. To appreciate why, recall that the image is loaded line by line into the SLAP. Hence, each pixel in a line is initially assigned to a processor of like index. To achieve the shuffled row-major distribution by interprocessor broadcasting would then require $\Omega(n^2)$ operations per processor for an $n \times n$ image, which would make it unsuitable for real-time applications. By contrast, the initial distribution requires only a single load operation per processor for each scan line. Accordingly, we assume this straightforward data distribution as a starting point for developing efficient algorithms for a selection of low and intermediate level image processing tasks.

1.2 Computational Model

The computational model used in this paper can be defined as follows. The SLAP architecture is a linear array of n processors, where each image to be processed is of size $n \times n$. Each processor is indexed in ascending order from 0 to $(n-1)$. After each scan line is received, each processor P_k latches in that value with the corresponding column index k. Each processor P_k is connected by a bidirectional communication link to processors $P_{(k-1)}$ and $P_{(k+1)}$, whenever they exist. Processors $P_{(k-1)}$ and $P_{(k+1)}$ are referred to as the *left* and *right* neighbors of processor P_k, respectively. The bandwidth of each communication link is assumed to be a word, where a word is defined to be $O(\log n)$ bits, and each processor can communicate a constant number of words to its immediate neighbors in a unit time. The processors operate together in a SIMD fashion, and each processor is a general purpose sequential processor with the capacity for conditional command execution and local address generation. In a unit of time, each processor can compute $O(1)$ basic logic or arithmetic operations. Finally, each processor has associated with it a random access memory which can hold $O(n)$ words. In a unit of time, each processor can access its local memory to read or write a word.

1.3 Problem Formulation

In this paper, the problem of developing algorithms for low level image processing is treated separately from that of intermediate level tasks. In the case of low level operations, it is assumed that we need to achieve the smallest possible latency consistent with an optimal running time. To this end, we require that the processing of a scan line begin immediately after it is received, and that the corresponding output line be generated in an amount of time independent of the image size. More formally, suppose that the required processing is based on neighborhoods of size $m \times m$. Then, we require that the output lines be generated at a rate of $O(m)$ operations per line after an initial delay of $O(m)$ lines. If an algorithm achieves this level of performance, then it runs in **real time**.

In the case of intermediate level operations, the entire image must be available for examination before an output line can be generated. Hence, we assume at the outset that the entire image is already stored in the local memories of the processors in the same way it is received, with each column stored at the processor of like index. As already noted, this straightforward input distribution requires only $O(n)$ time. Further, we require that any subsequent preprocessing be included in our time estimates. For these intermediate level operations, we call the algorithm **optimal** either if it runs in $O(n)$ time or it can be shown that no faster algorithm is possible.

1.4 Results

Our results can be summarized as follows. For low level operations, we develop real-time algorithms for block DFT, block DCT, convolution, template matching, shrinking, and expanding [20]. We also develop an algorithm for median filtering which runs in almost real-time at a cost of $O(m \log m)$ time per scan line and a latency of $\lfloor \frac{m}{2} \rfloor$ scan lines [20].

For intermediate level operations, we develop optimal algorithms for translation, histogram computation, scaling, and translation [20], although for the sake of brevity, we will omit discussion of the latter two algorithms. We also develop algorithms for labelling the connected components and for determining the convex hulls of multiple components which run in $O(n \log n)$ time and $O(n \log^2 n)$, respectively [20]. The complexities of these last two algorithms compare favorably with those of the existing algorithms, which respectively require $O(n)$ and $O(n \log n)$ time but which also assume the shuffled row-major distribution [19]. In addition, our algorithms are significantly simpler and easier to implement and are thus expected to perform better in practice.

2 Low Level Image Processing Operations

2.1 Median Filtering

Median filtering is the process of replacing each pixel of a given image by the median of the pixels contained in a window centered around that pixel. This filtering operation is useful in removing isolated lines or pixels while preserving spatial resolution. More specifically, let X be an input image of size $n \times n$ and let the window be of size $m \times m$, where m is assumed to be odd for convenience. The filtered image Y is defined by:

$$\begin{aligned} Y[j,k] \;=\; \text{median} \;\; & \{X[(j-r),(k-s)]:\; 0 \le (j-r),(k-s) \le (n-1) \\ & \qquad \text{and } -\lfloor \tfrac{m}{2} \rfloor \le r,s \le \lfloor \tfrac{m}{2} \rfloor\}, \end{aligned} \tag{1}$$

where $0 \le r,s \le (n-1)$. The straightforward approach to solving this problem on the SLAP is to compute the median independently for each output pixel. Specifically, to produce row j of the output image, each processor P_k first accumulates the m^2 pixel values necessary to compute $Y[j,k]$. The median of these m^2 values is then computed by the best known sequential algorithm in $\Theta(m^2)$ time. When the next scan line is received, each processor P_k updates its current set of pixel values by replacing the least recent m values by the current $(m-1)$ values located to either side of processor P_k. Clearly, this simple method yields the desired latency of $O(m)$ scan lines, but it also requires $\Omega(m^2)$ operations per scan line.

There is, however, a more efficient procedure that allows the median filtering of the image to be performed at the rate of $O(m \log m)$ operations per scan line with the same desired latency. This procedure is based on the observation that the window surrounding pixel $[j,k]$ and the window surrounding pixel $[(j+1),k]$ differ only by $2m$ pixels. This immediately suggests that we should store the m^2 values contained in the window about pixel $[j,k]$ in an appropriate data structure that allows us to (1) efficiently update the data structure to reflect the window about pixel $[(j+1),k)]$, and (2) quickly find the median of the values stored in the data structure. One such data structure is the order-statistic tree [21]. It allows us to dynamically (1) delete an element, (2) insert an element, or (3) locate the p^{th} smallest element (for any integer p) in time proportional to the logarithm of the tree size. Hence, by using this data structure, we can produce an algorithm which will perform median filtering in $O(m \log m)$ operations per scan line.

Assume for our real-time implementation of this operation that the input is the j^{th} row of the input image X. Initially, each processor P_k broadcasts the value of its input pixel

$X[j,k]$ to the set of $(m-1)$ processors $\{(k - \lfloor(\frac{m}{2})\rfloor), (k - \lfloor(\frac{m}{2})\rfloor + 1), ..., (k + \lfloor(\frac{m}{2})\rfloor)\}$ whose windows include $X[j,k]$. Since this essentially involves a left shift of the input scan line $\lfloor(\frac{m}{2})\rfloor$ positions followed by a right shift of $\lfloor(\frac{m}{2})\rfloor$ positions, it clearly can be accomplished in $O(m)$ time. Next, each processor P_k deletes the m oldest values in its order-statistic tree and then inserts both the value of $X[j,k]$ and the $m-1$ values received from its neighbors. As already noted, this can be accomplished in $O(m \log m)$ time. Finally, in $O(\log m)$ time, each processor P_k determines the median of its updated order-statistic tree and then outputs this result as the value of pixel $[(j - \lfloor(\frac{m}{2})\rfloor), k]$ of the output image. Thus, we have shown the following theorem:

Theorem 1: Given a SLAP with n processors, the median filtering operation on an $n \times n$ image with a window of size $m \times m$ can be performed in almost real-time at a cost of $O(m \log m)$ operations per scan line and a latency of $\lfloor(\frac{m}{2})\rfloor$ scan lines.

2.2 Block 2D-DFT and 2D-DCT

A block transformation of a given type on an image involves partitioning the image into non-overlapping blocks and then applying the desired transformation on each of the resulting blocks. Two of the most widely used transformations are the two dimensional discrete Fourier transform (2D-DFT) and the two dimensional discrete cosine transform (2D-DCT). In this section, we present an algorithm which can implement either transformation on the $m \times m$ blocks of an image matrix in real-time. Since the two problems are essentially analogous, we restrict our discussion to the task of computing the block 2D-DFT.

Let the $n \times n$ input image X be partitioned into non-overlapping $m \times m$ blocks, where we assume for convenience that m divides n evenly. If we denote one such block as $X_{(a,b)}$, where $a, b \in \{0, 1, \ldots, \frac{n}{m}\}$, then the 2D-DFT of $X_{(a,b)}$ is defined as:

$$Y_{(a,b)}[c,d] = \sum_{g=0}^{m-1} \sum_{h=0}^{m-1} X_{a,b}[g,h] e^{-\frac{i2\pi cg}{m}} e^{-\frac{i2\pi dh}{m}}, \tag{2}$$

where $i = \sqrt{-1}$ and $0 \le c, d \le (m-1)$. The straightforward approach to apply this transformation on the SLAP would be to independently evaluate the contribution of each input line to the computation of each coefficient, which would require $\Omega(m^2)$ time per scan line. However, because the 2D-DFT is a separable transform , we can rewrite its definition as follows:

$$\begin{aligned} Y_{(a,b)}[c,d] &= \textstyle\sum_{g=0}^{m-1} \left(\sum_{h=0}^{m-1} X_{a,b}[g,h] e^{-\frac{i2\pi dh}{m}} \right) e^{-\frac{i2\pi cg}{m}} \\ &= \textstyle\sum_{g=0}^{m-1} Y'_{a,b}[g,d] e^{-\frac{i2\pi cg}{m}} \end{aligned} \tag{3}$$

Notice that, for a given row g of the input block and a given column d of the output block, there is a single value of $Y'_{(a,b)}[d,g]$ which can be computed with $O(m)$ operations. Once $Y'_{a,b}[g,d]$ has been calculated, we can then evaluate the contribution of the g^{th} input row to the d^{th} output column with $O(m)$ operations. Thus, by taking advantage of its separability, we can compute the block DFT in real-time.

Assume for our real-time implementation of this operation that the input is the j^{th} row of the image X. Initially, each processor P_k broadcasts the value of its input pixel $X[j,k]$ to the set of $(m-1)$ processors $\{((\lfloor (k/m) \rfloor \times m), ((\lfloor (k/m) \rfloor \times m)+1), ..., ((\lfloor (k/m) \rfloor \times m)+m-1)\}$ which share the same block, which requires no more than $O(m)$ time. Following this, each processor P_k computes the $(k \bmod m)^{th}$ coefficient of the 1D-DFT of the $(j \bmod m)^{th}$ row of its block. Specifically, each processor P_k evaluates the expression:

$$Y'_k = \sum_{r=(\lfloor (k/m) \rfloor \times m)}^{((\lfloor (k/m) \rfloor \times m)+m-1)} X[j,r] e^{-\frac{i2\pi kr}{m}} \tag{4}$$

Clearly, this involves no more than $O(m)$ operations. Next, each processor P_k uses this result to update the column of partially computed 2D-DFT coefficients held in the array $Partial-DFT_k$ at its local memory. More precisely, for each value of c, where $0 \leq c \leq (m-1)$, each processor P_k performs the following operation:

$$Partial-DFT_k[c] = Partial-DFT_k[c] + Y'_k e^{-\frac{i2\pi cj}{m}} \tag{5}$$

Again, this clearly involves no more than $O(m)$ operations. When the last row of the block is processed, the values held in the array $Partial-DFT_k$ will be the $(k \bmod m)^{th}$ column of the of the 2D-DFT of the block just processed. Finally, each processor P_k outputs $Y_{((\lfloor (j/m) \rfloor -1), \lfloor (k/m) \rfloor)}[(j \bmod m), (k \bmod m)]$ as pixel $[(j-m), k]$ of the output image. Hence, we have the following theorem:

Theorem 2: Given a SLAP with n processors, the block DFT or DCT of an image of size $n \times n$ with each block of size $m \times m$ can be computed in real-time at a cost of $O(m)$ operations per scan line and a latency of m scan lines.

2.3 Convolution and Template Matching

Convolution and template matching are fundamental image processing operations that are computationally demanding. In this section, we present a method that performs the convolution of an image of size $n \times n$ with a kernel of size $m \times m$ in real-time - that is, at the rate of $O(m)$ operations per scan line. The same method can be extended to perform template matching within the same time bounds.

Given an image X of size $n \times n$ and a kernel W of size $m \times m$, the convolution of X with W is the image Y of size $(n+m-1) \times (n+m-1)$ defined by:

$$Y[j,k] = \sum_{r=0}^{m-1} \sum_{s=0}^{m-1} X[(j-r),(k-s)] \times W[r,s], \tag{6}$$

where $0 \leq j,k \leq (n+m-2)$. Here we are assuming implicitly that $X[(j-r),(k-s)]$ is equal to zero whenever $(j-r)$ or $(k-s)$ is not in the interval $[0,(n-1)]$.

A straightforward computation of the convolution would require $\Theta(n^2m^2)$ operations and therefore would require a minimum of $\Omega(m^2)$ operations per scan line on the SLAP. Suppose, instead, that we employ the overlap-and-add strategy [22]. Specifically, we partition the input image X into non-overlapping $m \times m$ blocks referred to as X' and indexed by (a, b). We then convolve each block $X'_{(a,b)}$ with the $m \times m$ kernel W to obtain the $(2m-1) \times (2m-1)$ block $Y'_{(a,b)}$. If we then let $a = \lfloor(\frac{j}{m})\rfloor$, $b = \lfloor(\frac{k}{m})\rfloor$, $c = j \bmod m$, and $d = k \bmod m$, then it is easy to verify that the value $Y[j,k]$ defined by (6) can now be obtained by simply adding the four entries $Y'_{a,b}[c,d]$, $Y'_{a-1,b-1}[c+m, d+m]$, $Y'_{a-1,b}[c+m, d]$, and $Y'_{a,b-1}[c, d+m]$ [22].

Up until now, nothing apparent has been gained computationally by redefining our matrix convolution problem as we have in terms of block convolution. However, we can now "zero pad" each block of X according to the following rule:

$$X''_{(a,b)}[c,d] = \begin{cases} X'_{(a,b)}[c,d] & \text{if } 0 \leq c, d < (m-1) \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where $0 \leq c, d \leq (2m-1)$. We can also "zero pad" our kernel W in an analogous manner to obtain W''. Having done this, we are now able to obtain the linear convolution of $X'_{(a,b)}$ and W by performing the circular convolution of their "zero-padded" counterparts, $X''_{(a,b)}$ and W''. The advantage of circular convolution lies in the fact that the circular convolution of two matrices can be obtained simply by finding their respective DFTs, multiplying the identically indexed elements of these two DFTs, and finally finding the IDFT of the resulting product matrix. This can be expressed more concisely as follows. Let $\bigodot$ denote circular convolution and $\bigotimes$ denote element-wise multiplication. Then:

$$X''_{(a,b)} \bigodot W'' = IDFT(\ DFT(X''_{(a,b)}) \bigotimes DFT(W'')\ ) \tag{8}$$

We have already demonstrated in the previous section that we can compute the block DFT of an $n \times n$ image with $m \times m$ blocks in $O(m)$ operations per scan line. Hence, we would also expect to be able to obtain the DFT of our $2m \times 2m$ "zero-padded" blocks in $O(m)$ time as well. Further, with the resulting DFT of these blocks distributed two columns per processor, it is clear that the element-wise multiplication of the DFT of $X''_{(a,b)}$ with the DFT of W would require no more than $O(m)$ operations per processor as well. Finally, since the computation of the IDFT is essentially analogous to the computation of the DFT, we would expect that a modification of our algorithm for finding the block DFT would also yield the IDFT of $(DFT(X''_{(a,b)}) \otimes DFT(W''))$ in $O(m)$ operations per input line. Of course, we would need to complete our computation of $(DFT(X''_{(a,b)}) \otimes DFT(W''))$ before we could begin our computation of its IDFT. Hence, we would actually overlap our computation of $(DFT(X''_{(a,b)})$ $\otimes$ $DFT(W''))$ not with the computation of the IDFT of $(DFT(X''_{(a,b)}) \otimes DFT(W''))$ but rather with the computation of the IDFT of $(DFT(X''_{((a-1),b)}) \otimes DFT(W''))$. Thus, we see that by converting our problem of convolution to one of block convolution and then taking advantage of the properties of Fourier transforms, we can reduce the cost of convolution from $O(m^2)$ operations per scan line to $O(m)$ operations per scan line. For a more detailed presentation of this algorithm, see [23].

Hence, we have the following theorem:

Theorem 3: Given a SLAP with n processors, the convolution of an image of size $n \times n$ with a kernel of size $m \times m$ can be computed in real-time at a cost of $O(m)$ operations per scan line and a latency of $2m$ scan lines. Similarly, the template matching problem can be solved at the same rate for a template of size $m \times m$.

2.4 Shrinking and Expanding

Given a positive integer m, the m-step shrinking of an $n \times n$ image X is the $n \times n$ image $S^{(m)}$ defined recursively as follows:

$$S^{(m)}[j,k] = \begin{cases} \min\{X[r,s] : -1 \leq (j-r),(k-s) \leq 1\} & \text{if } m = 1 \\ \min\{S^{(m-1)}[r,s] : -1 \leq (j-r),(k-s) \leq 1\} & \text{if } m > 1 \end{cases}, \tag{9}$$

where $0 \leq j,k \leq (n-1)$. Similarly, the m-step expansion of an $n \times n$ image X is the $n \times n$ image $E^{(m)}$ defined by replacing the minimum by the maximum in the above definition. It is easy to verify that the m-step shrinking and the m-step expansion simply involve the replacement of each pixel in X by the respective minimum or maximum of the $(2m+1)^2$ pixels contained in the $(2m+1) \times (2m+1)$ window centered about that pixel. Because of the essential similarity between the shrinking and expansion problems, we will only discuss an algorithm for the former.

Assume for our real-time algorithm that each processor P_k holds in its local memory a queue, referred to as $Queue_k$, which holds the previous $(2m+1)$ pixels input to that processor. Once a new pixel $X[j,k]$ is received at processor P_k, $Queue_k$ is updated and the minimum min_k of the values in $Queue_k$ is computed. Next, each processor P_k broadcasts min_k to the set of processors $\{(k-m),(k-m+1),...,(k+m)\}$ whose windows encompass column k. When this is completed, each processor has $(2m+1)$ values which represent the minima of each of the columns contained in the window centered at that processor. Each processor P_k then computes the smallest of these $(2m+1)$ values, which it then outputs as pixel $[(j-m),k]$ of the output image. Hence, we have the following theorem:

Theorem 4: Given a SLAP with n processors, the m-step shrinking or expansion of an image of size $n \times n$ can be computed in real-time at a cost of $O(m)$ operations per scan line and a latency of m scan lines.

3 Intermediate Level Image Processing Operations

3.1 Convex Hull

Given a set S of points distinguished by a common label, the convex hull is defined as the smallest convex polygon which contains all the points of the set. The extreme points of S are defined as the corners of this smallest polygon. In this section, we consider an $n \times n$ input image X in which each pixel can have any one of $O(n)$ possible labels. The pixels which share a particular label are said to belong to a set, even though they may be spatially unconnected to one another. For each of these $O(n)$ sets, we wish to determine the extreme points of its convex hull.

To do this, we present an algorithm which can compute these $O(n)$ convex hulls in $O(n \log^2 n)$ time. To simplify our presentation, we concentrate only on the problem of determining the upper hulls, since the task of finding the lower hulls is entirely analogous. Our divide-and-conquer algorithm first divides the input image into two subimages, one consisting of the leftmost $(\frac{n}{2})$ columns and the other consisting of the rightmost $(\frac{n}{2})$ columns.

For each of the $O(n)$ labels, the upper hulls of the two subimages are recursively computed in parallel, after which a novel strategy is used to merge the two upper hulls. The merging procedure consists of first concatenating the sequence of extreme points which define the right upper hull to the sequence of extreme points which define the left upper hull. Having done this, we then remove any subsequences of points $\{p_{l+1}, p_{l+2}, ..., p_{r-1}\}$ such that all these points lie on or below the line connecting the adjacent points p_l and p_r. In our algorithm, this is performed by making $O(\log n)$ left-to-right and right-to-left sweeps across the image for each label, each time eliminating a fraction of these non-extreme points. Extensive pipelining is used to insure that in $O(n)$ time a sweep can be completed for every one of the $O(n)$ labeled sets. In spite of the simplicity of our algorithm, its analysis requires a somewhat tricky geometric argument to establish that $O(\log n)$ sweeps are sufficient. Hence, we first present the algorithm and then provide a proof of its correctness.

3.1.1 Algorithm

We begin our algorithm with two preprocessing steps, each of which reduces the complexity of our task. In the first preprocessing step, each processor P_k sequentially examines the correspondingly indexed column of the input image X in its local memory. For each label it encounters, it retains the uppermost occurrence of that label as a potential candidate point for the upper hull of that set. In the second preprocessing step, for each label, we make a left-to-right sweep across the processor array. As we pass each processor P_k, we identify, for the candidate point (if any) at that processor, the candidate point from processors 0 through k which has the maximum row index. Similarly, for each label, we also make a right-to-left sweep across the processor array. Again, as we pass each processor P_k, we identify for the candidate point (if any) at that processor the candidate point from processors k through $(n-1)$ which has the maximum row index. By pipelining the sweeps for each of the $O(n)$ labels, the whole process can be completed in $O(n)$ time. Each processor P_k then examines each of its candidate points to see if it lies below the line connecting its respective left and right maxima. If it does, then that point can be eliminated as a possible candidate point. For each label, the sequence of candidate points left after these two preprocessing steps actually consists of two characteristic subsequences, one of which may be empty. The first subsequence is monotonically increasing, and the other subsequence is monotonically decreasing. The significance of this property will become apparent when we discuss our proof.

The algorithm for computing the extreme points of the upper hulls proceeds as follows. Divide the $n \times n$ input image X into two subimages X_1 and X_2, where X_1 consists of columns 0 through $(\frac{n}{2}-1)$ and X_2 consists of columns $(\frac{n}{2})$ through $(n-1)$. For each of the $O(n)$ labels, we recursively compute in parallel UH_1 and UH_2, which are respectively the upper hulls of the sets of points with that label found in X_1 and X_2. When this computation is completed, UH_1 and UH_2 are represented for a given label by the remaining candidate points at columns 0 through $(\frac{n}{2}-1)$ and columns $(\frac{n}{2})$ through $(n-1)$, respectively.

To merge UH_1 and UH_2 and thereby obtain the completed upper hull UH, we proceed as follows. First, we compute for each candidate point p_i the coordinates of its immediate successor $S(p_i)$. For each label, this can be accomplished by making a single right-to-left pass across the processor array, carrying along the coordinates of the most recently encountered candidate point with that label. Again, by pipelining these sweeps for each of the $O(n)$ labels, the whole process can be completed in $O(n)$ time. Next, for $O(\log n)$ repetitions, we perform the following computation. For each label, we make a left-to-right pass across the processor array, again carrying along the coordinates of the most recently encountered candidate point with that label. Hence, when we arrive at a candidate point p_i, we have available the coordinates of its current predecessor $P(p_i)$. Denote the angle formed by the points $P(p_i)$, p_i, and $S(p_i)$ which opens away from the interior of the convex hull as $\angle P(p_i), p_i, S(p_i)$. At each candidate point p_i, we check to see if $\angle P(p_i), p_i, S(p_i) \leq 180°$. This corresponds to asking whether p_i lies on or below the line segment connecting $P(p_i)$ to $S(p_i)$. If it does, then we eliminate this point from consideration and we carry along the coordinates of $P(p_i)$ as the new value of the predecessor of $S(p_i)$. When we complete this left-to-right pass, we then begin an analogous right-to-left pass back across the processor array. Now, however, we have to be concerned that the deletions of the previous left-to-right pass may have left some successor values outdated. Hence, we bring along to each candidate point p_i the coordinates of its immediate successor. Pipelining again insures that we can complete these two passes for all $O(n)$ labels in $O(n)$ time. And, after $O(\log n)$ repetitions, the candidate points remaining for each set will be the extreme points of the upper hull of that set.

3.1.2 Proof

We have made the claim that $O(n \log^2 n)$ time is sufficient to determine the extreme points of the convex hulls of $O(n)$ arbitrary sets. The basis of this claim is our contention that $O(\log n)$ passes across the processor array are sufficient to merge any two arbitrary upper hulls. We will first prove this latter claim, and then we will demonstrate that the overall running time follows.

The proof of the upper limit on the running time of our merging algorithm is based on a geometric argument, and therefore it is most clearly presented by a diagram such as that shown in Figure 1. Figure 1 illustrates the process of merging two arbitrary upper hulls, UH_1 and UH_2, where UH_1 is defined by the subsequence of extreme points $\{p_1, p_2, ..., p_j\}$ and UH_2 is defined by the subsequence of extreme points $\{p_{(j+1)}, p_{(j+2)}, ..., p_k\}$. The order of the points in each subsequence corresponds to the relative order of their column indices. We assume that UH_1 spans some portion of columns 0 through $(\frac{n}{2} - 1)$ and UH_2 spans some portion of columns $(\frac{n}{2})$ through $(n-1)$. As such, each upper hull can be defined by up to $(\frac{n}{2})$ extreme points. According to the way we have drawn UH_1 and UH_2 in Figure 1, the upper hull UH resulting from merging UH_1 and UH_2 will be the line segment $\overline{p_1 p_k}$.

For the sake of clarity, we have allowed a number of distortions in Figure 1. First, we have represented UH_1 and UH_2 as continuous curves rather than as concatenations of line segments. Second, we have allowed the angle $\angle p_{(j-1)}, p_j, p_{(j+1)}$ opening away from the interior of the convex hull to be less than $90°$, when in fact our preprocessing makes this impossible. Finally, we have drawn UH_1 and UH_2 so that merging will eliminate the entire subsequence $\{p_2, p_3, ..., p_{(k-1)}\}$. Yet, this subsequence is composed of two smaller subsequences $\{p_2, p_3, ..., p_j\}$ and $\{p_{(j+1)}, p_{(j+2)}, ..., p_{(k-1)}\}$, one of which is shown as being monotonically decreasing and the other of which is shown as being monotonically increasing. To understand the difficulty with this, recall that our preprocessing procedure insures that the original se-

Figure 1: Merging of Two Upper Hulls

quence of candidate points is monotonically increasing up to one or two maximum points and then monotonically decreasing thereafter. Clearly, these maximum point(s) must also be extreme points of the completed convex hull, implying that only subsequences of candidate points which lie either to the left or right can qualify for elimination. Therefore, any subsequence of non-extreme candidate points which need to be eliminated must be either monotonically increasing or decreasing, and so too with $\{p_2, p_3, ..., p_{(k-1)}\}$. However, a situation such as that pictured in Figure 1 could still arise if we allow ourselves to rotate the image so that the line segment $\overline{p_1 p_k}$ is parallel to the horizontal axis of the image grid. We permit all these changes described because they enhance the clarity of our presentation without affecting the validity of our proof.

Consider a arbitrary left-to-right pass across the image as described in our algorithm. This sweep extends a tangent to UH_2 from the second-to- last point remaining in UH_1, thereby eliminating all the intervening points which fall below this line. Assume that an arbitrary number of passes have been made across the processor array, each time eliminating points from the end of UH_1, and that now p_L is the last point in the remaining sequence. The next left-to-right pass extends the line segment $\overline{P(p_L)p_T}$, as shown in Figure 1. Following this, the succeeding right-to-left pass draws the tangent $\overline{S(p_T)p_A}$. Two facts concerning the point p_A must hold. First, p_A must lie to the left of the line segment $\overline{P(p_L)G}$, the perpendicular to $\overline{p_1 p_k}$ passing through point $P(p_L)$. To see why, recall that any subsequence of candidate points which requires elimination must be either monotonically increasing or monotonically decreasing. As a consequence, the angle formed by any three points p_u, p_v, and p_w such that $u < v < w$ must be greater than $90°$, which in turn means that the perpendicular to $\overline{p_1 p_k}$ drawn through $P(p_L)$ must divide the subsequence into exactly two parts. Thus, every point remaining from $UH1$ aside from p_L and $P(p_L)$ must lie to the left of the line segment $\overline{P(p_L)G}$, and so too with p_A. We also require that the point p_A lie on or above the line segment $\overline{p_T C}$, which is the line segment parallel to $\overline{p_1, p_k}$ passing through p_T. This is guaranteed by the fact that the angle formed by any three consecutive points on the upper hull must be greater than 180°. Hence, if p_u, p_v, and p_w are any three consecutive points which determine UH_1, the slope of $\overline{p_v p_w}$ defined with respect to $\overline{P(p_L)G}$ must be steeper than $\overline{p_u p_v}$. Consequently, the slope of $\overline{S(p_T)p_A}$ defined with respect to $\overline{P(p_L)G}$ must be steeper than $\overline{p_1, p_k}$, and therefore $\overline{S(p_T)p_A}$ must lie above $\overline{p_T C}$.

Following the right-to-left pass that has drawn the tangent $\overline{S(p_T)p_A}$, the succeeding left-to-right pass draws the tangent $\overline{P(p_A)p_B}$. Notice that for the same reason that point p_A must lie on or above the line segment $\overline{p_T C}$, point p_B must lie on or above the line segment $\overline{p_A H}$, which is the line segment parallel to $\overline{p_1, p_k}$ passing through p_A. Moreover, point p_B must lie to the right of the line passing through points $P(p_L)$ and p_T, since the angle formed by any three consecutive points on the upper hull must be greater than 180°. Finally, if we define point F to be the point on the line passing through points $P(p_L)$ and p_T which is the same distance from p_T as $P(p_L)$, then we can divide the region to the right of this line into two halves as indicated in Figure 1. This enables us to make the following two observations:

(1) If p_B lies in region 1, then the projection of $\overline{P(p_A)p_B}$ on $\overline{p_1 p_k}$ must be greater than or equal to the projection of $\overline{P(p_L)F}$. Since by definition the projection of $\overline{P(p_L)F}$ is exactly twice the projection of $\overline{P(p_L)p_T}$, it follows that the projection of the second tangent $\overline{P(p_A)p_B}$ must be at least twice the projection of the previous tangent $\overline{P(p_L)p_T}$.

(2) Since point p_A must lie above $\overline{p_T C}$ and to the left of $\overline{P(p_L)G}$, it follows that the slope of $\overline{P(p_A)p_B}$ defined with respect to $\overline{P(p_L)G}$ must be less than the slope of $\overline{Dp_B}$. If p_B lies in region 2, then the slope of $\overline{Dp_B}$ must be less than the slope of $\overline{DF}$. Since by definition the slope of $\overline{DF}$ is exactly half the slope of $\overline{P(p_L)F}$, it follows that the slope of the second tangent $\overline{P(p_A)p_B}$ must be less than half the slope of the previous tangent $\overline{P(p_L)p_T}$.

Hence, we can conclude that the tangent drawn by each left-to-right pass is either half the slope or double the projection of the tangent drawn by the previous left-to-right pass.

Because the image is digitalized, it can be easily shown that the smallest possible nonzero projection of a tangent on either $\overline{p_1 p_k}$ or its perpendicular is $\Omega(\frac{1}{n})$. Of course, the maximum possible projection of $\overline{p_u p_v}$ on either $\overline{p_1 p_k}$ or its perpendicular is obviously $O(n)$. It follows from this that the maximum and minimum possible slopes of $\overline{p_u p_v}$ defined with respect to $\overline{p_1 p_k}$ are $O(n^2)$ and $\Omega(\frac{1}{n^2})$, repectively. And, therefore, since the tangent drawn by each left-to-right pass is either half the slope or double the projection of the tangent drawn by the previous left-to-right pass, it follows that $O(\log n)$ such passes are sufficient.

To compute the upper bound on the overall running time for determining the $O(n)$ possible convex hulls, we note that the solution of the merging problem for two $n \times \frac{p}{2}$ subimages $(2 \leq p < n)$ still requires $\Theta(n \log n)$time. Hence, the upper bound on the running time for our divide-and-conquer algorithm is governed by the following recurrence:

$$T(n,p) \leq \begin{cases} O(n \log n) & \text{if } p = 2 \\ T(n, \frac{p}{2}) + O(n \log n) & \text{if } p > 2 \end{cases} \tag{10}$$

whose solution is $O(n \log^2 n)$. Hence, we have the shown the following theorem:

Theorem 5: Assume that we have a SLAP with n processors and an input image of size $n \times n$ distributed one column per processor. If we also assume that each pixel is labelled with one of $O(n)$ possible labels, then we can determine the extreme points of the convex hulls of all these labeled sets in $O(n \log^2 n)$ time.

We can also use this algorithm to solve another problem involving convex hulls. Consider an $n \times n$ image in which each pixel can belong to any one of $O(n^2)$ sets distinguished by a common label. If we require that all the pixels which belong to a particular set must form a connected component, then with appropriate preprocessing we can use our algorithm to find the extreme points of the convex hulls of all $O(n^2)$ sets in $O(n \log^2 n)$ time. To justify this claim, consider an arbitrary vertical line that divides the image into two parts. Because of our requirement that the members of a particular set all belong to a connected component, no more than n sets can have members on both sides of this arbitrary line. Hence, when our algorithm calls for us to make a pass across the processor array for a given set, we obviously only need to make a pass across this arbitrary line if this set is one of those n sets. Therefore, as we move from one processor to another, the coordinates of only $O(n)$ candidate points and their labels need to be carried along, and hence $O(n \log^2 n)$ time suffices for this case as well. Hence, we also have the following theorem:

Theorem 6: Assume that we have a SLAP with n processors and an input image of size $n \times n$ distributed one column per processor. If we also assume that each pixel is labelled with one of $O(n^2)$ possible labels with the restriction that all those pixels which share a particular label must form a connected component, then we can determine the extreme points of the convex hulls of all these labeled sets in $O(n \log^2 n)$ time.

3.2 Connected Components

Given a binary image X, two pixels $p_{(j_0,k_0)}$ and $p_{(j_1,k_1)}$ are called *neighbors* if $-1 \leq (j_0 - j_1), (k_0 - k_1) \leq 1$. In other words, two pixels are said to be neighbors if they are adjacent horizontally, vertically, or diagonally. Two pixels $p_{(j_1,k_1)}$ and $p_{(j_i,k_i)}$ are then said to be *connected* if there exists a sequence of pixels $p_{(j_h,k_h)}$, $1 \leq h \leq i$, sharing some property such that $p_{(j_{(h-1)},k_{(h-1)})}$ is a neighbor of $p_{(j_h,k_h)}$. In this section, we present an algorithm which labels the connected components of an $n \times n$ input image on the SLAP in $O(n \log n)$ time.

Our algorithm for finding the connected components employs the divide-and-conquer strategy. Assume that each processor P_k holds the k^{th} column of the input image. We divide the $n \times n$ input image X into two subimages X_1 and X_2, where X_1 consists of columns 0 through $(\frac{n}{2} - 1)$ and X_2 consists of columns $(\frac{n}{2})$ through $(n-1)$ Recursively and in parallel, we identify and label the connected components of the two subimages X_1 and X_2. When this is completed, we identify and merge those connected components which span columns $(\frac{n}{2} - 1)$ and $(\frac{n}{2})$. This can be easily done using existing sequential algorithms in $O(n)$ time by either processor $P_{(\frac{n}{2}-1)}$ or processor $P_{(\frac{n}{2})}$. We then broadcast any label changes across the two subimages. Since there can be no more than $O(n)$ connected components spanning these two columns and hence no more than $O(n)$ label changes, this too can be accomplished by pipelining in $O(n)$ time. Clearly, then, the whole merging operation can be completed in $O(n)$ time.

To compute the upper bound on the running time for labeling the connected components, we note that the solution of the merging problem for two $n \times \frac{p}{2}$ subimages $(2 \leq p < n)$ still requires $\Theta(n)$ time. Hence, the upper bound on the running time for our divide-and-conquer algorithm is governed by the following recurrence:

$$T(n,p) \leq \begin{cases} O(n) & \text{if } p = 2 \\ T(n, \frac{p}{2}) + O(n) & \text{if } p > 2 \end{cases} \tag{11}$$

whose solution is $O(n \log n)$. Hence, we have the following theorem:

Theorem 7: Given a SLAP with n processors and an input image of size $n \times n$ distributed one column per processor, the connected components of this image can be labeled in $O(n \log n)$ time.

3.3 Translation

Translation is the process of mapping each pixel at location $[j,k]$ in the $n \times n$ input image X to a new position $[j', k']$ in the $n \times n$ output image Y such that:

$$\begin{aligned} j' &= ((j + a) \bmod n) \\ k' &= ((k + b) \bmod n), \end{aligned} \tag{12}$$

where $j', k' \in \{0, 1, 2, \ldots, (n-1)\}$. We assume for simplicity that a and b are integers and that $|a|, |b| \leq (n/2)$, since the cases of when either $|a| > (n/2)$ or $|b| > (n/2)$ can be reformulated so that $|a|, |b| \leq (n/2)$.

The algorithm proceeds as follows. Initially, we shift each row $|b|$ positions to the right (if $b > 0$) or left (if $b < 0$), which correctly relocates all but $|b|$ columns. Following this, we wrap around the remaining $|b|$ columns to the other end of the processor array by pipelining their column-by-column broadcast. The task of translating the pixels within a particular column by $|a|$ positions is easily accommodated by insuring that as each pixel reaches the correct processor, it is stored at the properly shifted memory location.

It is straightforward to verify that the upper bound on the running time of this algorithm is $O(n|b|)$. For comparison, a lower bound can be derived by noting that the problem requires us to move $(n - |b|)$ columns of pixels a distance of $|b|$ processors and $|b|$ columns a distance of $(n - |b|)$ processors. Since each processor can perform a constant number transfers per unit time and there are n such processors, it follows that the lower bound is $\Omega(n|b|)$. Hence, we have the following theorem:

Theorem 8: Given a SLAP with n processors and an input image of size $n \times n$ distributed one column per processor, the translation of this image by a distance a in the vertical direction and a distance b in the horizontal direction can be completed in $\Theta(n|b|)$ time. This bound is optimal.

3.4 Histogram Computation

The histogram of an $n \times n$ input image X is a computation which determines the relative frequency of occurrence of the various possible pixel values in the image. To develop an algorithm for this, we assume for simplicity that the possible pixel intensities are constrained to integer values ranging from 0 up to $(m-1)$, where nothing is assumed about the relative sizes of m and n.

The algorithm proceeds as follows. First, each processor P_k computes the histogram of the k^{th} column of the input image. To do this, each processor simply examines each of these n pixels and increments the entry in an array called H_k whose index corresponds to the pixel value. Next, we compute the image matrix histogram by adding together the identically indexed values in each of the local histograms H_k. This simply requires m right-to-left passes across the processor array, which will then leave the completed histogram values in the array H_0 kept at processor P_0. If we pipeline these m passes, then the whole procedure can be completed in $O(m+n)$ time, which is clearly optimal. Hence, we have the following theorem:

Theorem 9: We are given a SLAP with n processors and an input image of size $n \times n$ distributed one column per processor. If we assume that the possible pixel values are constrained to integer values in the interval $[0,(m-1)]$, then the histogram of this image can be computed in $O(n+m)$ time.

4 Conclusion

In this paper we have presented efficient algorithms for the SLAP for a variety of low and intermediate level image processing tasks. For the low level operations, our algorithms run in real-time, which is clearly the best that can be achieved on this model. For the intermediate level operations, our algorithms are either optimal or compare favorably with the existing algorithms. Moreover, our algorithms achieve their performance without assuming that the input image is already partitioned according to the shuffled row-major distribution. Taken together, our results suggest that the SLAP is a promising architecture for real-time image processing.

References

[1] K.E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transactions on Computers*, (1980), pp. 836-840.

[2] T.J. Fountain, K.N. Mathews, and M.J.B. Duff, "The CLIP7A Image Processor", *IEEE Trans. on Pattern Anal. Machine Intell.*, (1980), pp. 310-319.

[3] J.R. Nickolls,"The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer," in *IEEE Digest of Papers-Compcon*, IEEE Computer Society Press, (1990), pp. 25-28.

[4] S.F. Reddaway, "DAP - A Distributed Processor Array", *First Ann. Symp. Comput. Architect.*, (1973), pp. 61-65.

[5] Eugene L. Cloud, "The Geometric Arithmetic Parallel Processor", *Proc. of 2nd Symposium on the Frontiers of Massively Parallel Computation*, (1988), pp. 373-381.

[6] V. Cantoni and S. Levialdi, "PAPIA", in *Parallel Computer Vision*, L. Uhr ed., Academic Press, (1987), pp. 3-14.

[7] D. Schaefer et al., "The GAM Pyramid", in *Parallel Computer Vision*, L. Uhr ed., Academic Press, (1987), pp. 15-42.

[8] C.C. Weems, S.P. Levitan, A.R. Hanson, E.M. Riseman, D.B. Shu, and J.G. Nash, "The Image Understanding Architecture", *Int. J. Computer Vision*, (1989), pp. 251-282.

[9] R.M. Lea, "The Asp: A Cost-Effective Parallel Microcomputer", *IEEE Micro*, (1988), pp. 10-29.

[10] M. Sharma, J.H. Patel, and N. Ahuja, "NETRA: An Architecture for a Large Scale Multiprocessor Vision System", in *Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, (1985), pp. 92-98.

[11] A.I. Fisher and P.T. Highnam, "Real-Time Image Processing on Scan Line Array Processors", *IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management*, (1985), pp. 484-489.

[12] A.I. Fisher, "Scan Line Array Processors for Image Computations", *International Conference on Computer Architecture*, (1986), pp. 338-345.

[13] D. Chin et. al., "The Princeton Engine: A Real-Time Video System Simulator", *IEEE Trans. on Consumer Electronics*, (May 1988), pp. 285-297.

[14] S. Knight et al., "The Sarnoff Engine: A Massively Parallel Computer for High Definition System Simulation", *Proceedings of ApplicationSpecific Array Processors*, (1992), pp. 342-357.

[15] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers", *IEEE Trans. on Computers*, (1978), pp. 84-87.

[16] I.V. Ramakrishnan and P.J. Varman, "Modular Matrix Multiplication on a Linear Array", *IEEE Trans. on Computers*, (1984), pp. 952-958.

[17] A.L. Fisher and P.T. Highnam, "Computing the Hough Transform on a Scan-Line Array Processor", *IEEE Trans. Pattern Anal. Machine Intell."*, (1989), pp. 262-265.

[18] K. Doshi and P. Varman, "Optimal Graph Algorithms on a Fixed-Size Linear Array", *IEEE Transactions on Computers*, (1987), pp. 460-470.

[19] H.M. Alnuweiri and V.K. Prasanna , "Optimal Geometric Algorithms for Digitized Images on Fixed-Size Linear Arrays and Scan-Line Arrays", *Distributed Computing*, (1991), pp. 55-65.

[20] A.K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, (1989).

[21] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw Hill, (1991), 281-286.

[22] A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, (1989).

[23] D.R. Helman, "Efficient Image Processing Algorithms on the Scan Line Array Processor", M.S. Thesis, University of Maryland, College Park, Maryland, (1993).