Specialized Computer Architectures for Robotics and Automation,
(Ed.) J. Graham, Gordon and Breach Science Publ., 1987, pp. 103-149.

COMPUTER ARCHITECTURES
FOR
ROBOTICS AND AUTOMATION

Edited by James H. Graham

Chapter 5

# ARCHITECTURES FOR ROBOT VISION*

T.N. Mudge
and
T.S. Abdel-Rahman

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

## INTRODUCTION

Interest in the area of computer processing of visual data has increased considerably over the past decade. This has led to an increasing number of applications for computers in that area. The applications vary from Robot Vision in industrial environments to Tomography in medical environments. It is safe to predict that the range of applications will continue to expand in the near future to cover many other application areas and give a wider scope to already existing applications. A partial list of applications include the following [1]:

- Automation of Industrial Processes.
  - Object acquisition together with a robot arm ('bin of parts problem').

* This work was supported in part by the Materials Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), United States Air Force, Wright-Patterson AFB, Ohio 45433-6503.

- — Automatic tool guidance.
- — Visual feedback for automatic assembly and repair.
- — VLSI (Very Large Scale Intergration) circuit inspection and checking.
- — Inspection of printed circuit boards.
- — Screening and inspecting plant samples.
- — Inspection of castings for impurities and fractions.

- • Medical Applications.
  - — Tomography.
  - — Enhancement and analysis of X-ray images.
  - — Blood cell recognition and counting.
  - — Tissue and chromosome analysis.
  - — Aiding the blind.
- • Remote Sensing.
  - — Cartography.
  - — Monitoring traffic.
  - — Exploration of remote and hostile areas.
- • Military Applications.
  - — Tracking moving objects.
  - — Automatic navigation.
  - — Target acquisition and range finding.

There are several disciplines that are concerned with the processing of visual data or images. The primary ones are: *Image Processing*, which is concerned with the transmission, storage, enhancement and restoration of images; *Pattern Recognition*, which is concerned with classifying regions of the input image as one of a (usually small) set of possibilities; and *Image Understanding*, which is aimed at the construction of a rich set of descriptors of images to facilitate the interpretation and the understanding of those images [1]. Secondary disciplines include: *Computer Graphics* and *Computer Aided Design*. Systems that process images benefit from the above disciplines to varying degrees depending on the specific nature of the application at hand.

The objective of this chapter is to study special purpose architectures for Robot Vision. This will be accomplished in two steps. First, algorithms for Robot Vision are discussed and a set of benchmarks of typical algorithms for Robot Vision is presented. Second, various classes of special purpose architectures for Robot Vision are

discussed. The set of benchmarks is then used to evaluate the performance of the various architectures and illustrate their basic characteristics.

The remainder of the chapter is organized as five sections: the rationale behind special architectures for Robot Vision; basic terminology that is used throughout this chapter; algorithms for Robot Vision including the set of benchmark algorithms; special computer architectures for Robot Vision and their performance on the benchmarks; finally, conclusions and final remarks.

## RATIONAL FOR SPECIAL ARCHITECTURES

Robot Vision applications require very high computational power that is beyond the capabilities of modern conventional computers. This is principally due to the massive amounts of data that has to be processed in such applications. Images, typically represented as two dimensional arrays of $M^2$ pixels (picture elements), may have to be processed at the rate of 30 images per second to meet real-time video rate requirements. Typical values of $M$ range from 256 to 4096. In some applications, however, $M$ can be as large as $10^4$. Each pixel in the image can have from 1 bit to 24 bits of data representing a gray level value or a color intensity.

To illustrate the computational effort needed to process images, Table 1 shows the processing time for a $512 \times 512 \times 1$ byte image on a conventional computer such as the VAX 11/780 supermini, which is capable of 1 MIPS (Million Instructions Per Second). The first column shows the number of operations performed on each pixel in the image. The second column shows the approximate number of instructions needed to process the image. An average of four instructions per operation is assumed. The third column gives the execution time for the image. The numbers indicate that a conventional computer would fall well behind the required computational effort. Larger images require even longer processing times. Furthermore, 30 images per second must be processed if video rates are required.

Hence, it becomes necessary to employ special purpose machines

| Ops per pixel | Total No. of instrs. | Time for a VAX |
|---|---|---|
| $10^2$ | $10^8$ | 100 sec |
| $10^3$ | $10^9$ | 17 min |
| $10^4$ | $10^{10}$ | 2.78 hr |

Table 1 Processing times for a 512 × 512 × 1 byte image on a VAX 11/780

that have sufficient processing capabilities to provide the necessary speedup over conventional computers. This can be achieved through three means:

1. Using a large number of processors working in parallel. This is referred to as *Parallel Processing*.
2. Using faster computer hardware technology.
3. Using better programming through optimizing compilers and new algorithms.

Parallel processing is the basic means for providing computational power for Robot Vision. Newer computer hardware technologies, while providing high levels of integration that roughly quadruple every two years, have only been able to increase speed by a factor of 10 in the past decade [2]. Better programming cannot provide a substantial speedup in Robot Vision applications due to the simple but repetitive nature of the operations involved.

Furthermore, the nature of image data and the nature of Robot Vision algorithms (as will be clear later) make parallel processing very attractive. Many Robot Vision algorithms involve operations that transform the value of each pixel in the image based on the values of surrounding pixels. This locality in operations suggests that image data be decomposed into a large number of subsets that can be processed in parallel. Hence, it is not surprising that an architectural characteristic of most special processors for Robot Vision is the employment of parallel processing to achieve the necessary computational power.

## TERMINOLOGY

An image, a frame or a picture, $P$, is a two dimensional array of pixels, $P(i,j)$, $i,j = 1,\ldots, M$. The term $P(i,j)$ can stand for the name of the pixel at $(i,j)$ or the gray level value at $(i,j)$. The gray level value is typically an integer in the range $0,\ldots,255$. $P(0,0)$ is located at the top left corner of the array. The index $i$ runs in the negative $y$-direction. The index $j$ runs in the $x$-direction. This is shown in Figure 1.

A neighborhood of a pixel $P(i,j)$, denoted by $N(i,j)$, is a set of contiguous pixels that include $P(i,j)$. Some of the common neighborhoods are:

- $m \times m$ neighborhood, ($m$ odd). Defined as

$$N(i,j) = \bigcup_{\alpha,\beta} P(i+\alpha, j+\beta) \qquad \alpha,\beta = -\left\lfloor \frac{m}{2} \right\rfloor, \ldots, \left\lfloor \frac{m}{2} \right\rfloor.$$

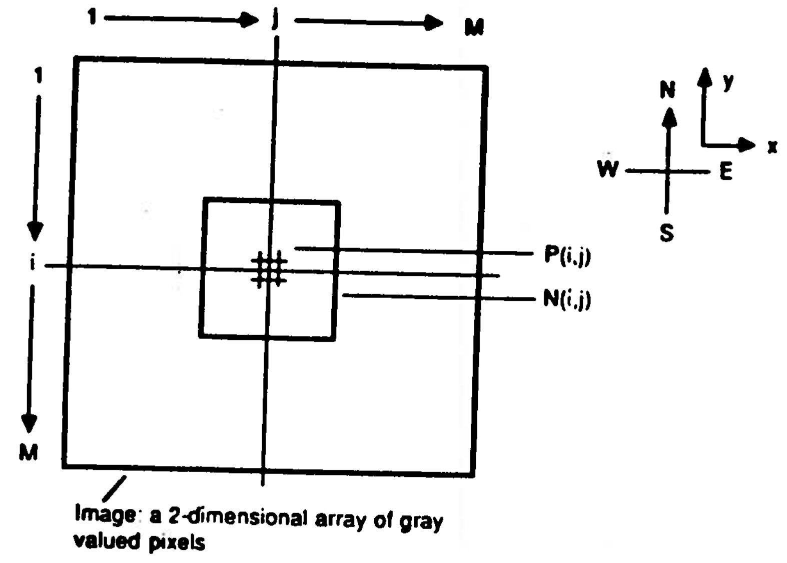


FIGURE 1 An image $P(i,j)$.

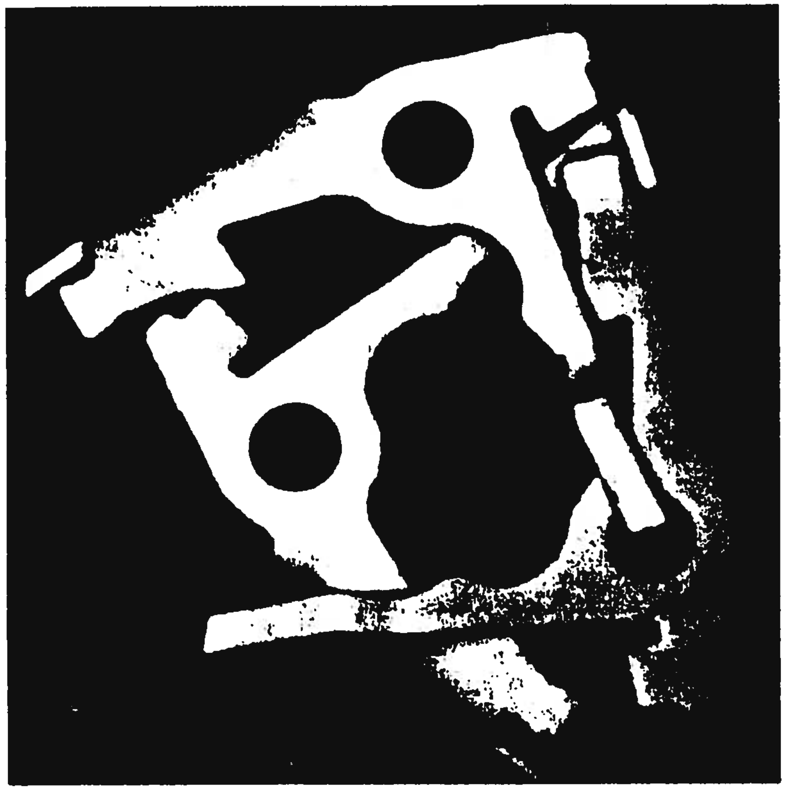


FIGURE 2 The heap-of-parts image.

The $3 \times 3$ neighborhood is common to many Robot Vision algorithms.

- $m \times 1$ neighborhood, defined as

$$N(i,j) = \bigcup_{\alpha} P(i + \alpha, j) \qquad \alpha = -\left\lfloor \frac{m}{2} \right\rfloor, \ldots, \left\lfloor \frac{m}{2} \right\rfloor.$$

- $1 \times m$ neighborhood, defined as

$$N(i,j) = \bigcup_{\beta} P(i, j + \beta) \qquad \beta = -\left\lfloor \frac{m}{2} \right\rfloor, \ldots, \left\lfloor \frac{m}{2} \right\rfloor.$$

Throughout this chapter, the image shown in Figure 2 will be used to illustrate Robot Vision algorithms. The image contains a heap of industrial parts.

## ALGORITHMS FOR ROBOT VISION

The processing of images for Robot Vision proceeds in the three major stages. In the first stage, referred to as the *low level processing* stage, the image is enhanced and features in the image are detected. Algorithms that perform these functions are referred to as *low level algorithms*. Low level processing draws most of its techniques from the field of image processing. Hence, the low level processing stage may be considered the stage of image processing. In the second stage, referred to as the *intermediate level processing* stage, the features detected in the first stage are extracted from the image. Algorithms that perform such functions are referred to *intermediate level algorithms*. In the third and final stage, referred to as the *high level processing* stage, the extracted features from the image in the second stage are classified and analyzed. Algorithms that perform these operations are referred to as *high level algorithms*. High level processing draws most of its techniques from the fields of pattern recognition and image understanding. Hence, the high level processing stage may be considered the stage of image analysis and understanding.

The above mentioned stages of processing will be referred to as the *Robot Vision System (RVS)*. The RVS is depicted in Figure 3.

In this section, the basic characteristics of each processing stage and its algorithms are discussed. Benchmarks of typical algorithms from each stage are also presented.

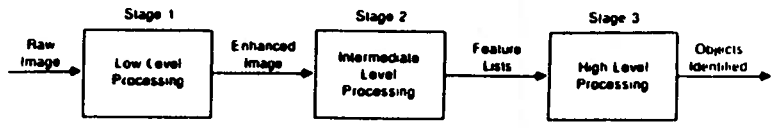


FIGURE 3 The Robot Vision system.

## Low Level Processing

Low level processing generally involves enhancement, restoration, noise removal and feature detection operations. This stage of processing is characterized by:

1. The image is represented as a two dimensional array of pixels. Each pixel represents the gray level value at its coordinates. This representation of the image, which is referred to as a *low level representation*, can be characterized in the following manner:
   (a) The spatial information regarding pixel location is implicit in the representation of the array. That is, no explicit address information is stored.
   (b) The classification of pixels is explicit in the representation through the values of the pixels in the array.
   (c) The features in the image are implicit in the representation through the relationships among the pixels.
2. A set of deterministic operations is applied to each pixel in the image. As a consequence, equal size areas of the image take equal amounts of processing times. Furthermore, because of the deterministic nature of the operations, no data dependent decisions are made during run-time. These operations basically are of four types[3]:
   (a) *Input/Output Operations*: for human interaction and image storage/retrieval.
   (b) *Context-Free Operations*: point-wise operations on single or multiple images. Examples include histogram generation and general tonal mapping.
   (c) *Context-Dependent Operations*: the value of a pixel is modified based on the values of the pixels in its context or neighborhood. Hence, these operations are also referred to as neighborhood operations. Context dependent operations form the majority of low level operations. Examples include: edge and line detection, noise removal and filtering.
   (d) *Global Transformation Operations*: the image is transformed by Fourier, Cosine, Hadamard and similar transforms.

The following are the set of low level algorithms used in our benchmark. They illustrate the general characteristics described above for low level algorithms.

### *Convolving with an FIR function*

In this algorithm, the input image is convolved with a Finite Impulse response (FIR) function to give the output image. The FIR function is a matrix $K(\alpha, \beta)$ of constant coefficients of dimensions $m \times m$. This FIR matrix is also referred to as the *kernel*. The kernel is moved across the image in one pixel steps to implement the convolution function. In each step, the pixel $P(i,j)$ coinciding with the center of the kernel is replaced by $Q(i,j)$, where

$$Q(i,j) = \sum_{\alpha}\sum_{\beta} P(i+\alpha, j+\beta)K(\alpha, \beta).$$

Convolving the image with a FIR function falls into the type of context dependent operations. The kernel $K(i,j)$ can be viewed to coincide with the $m \times m$ neighborhood for the pixel $P(i,j)$. The value of $P(i,j)$ is then replaced by a new value, $Q(i,j)$, which is a function, linear in this case, of the pixels in the neighborhood. This is shown in Figure 4.

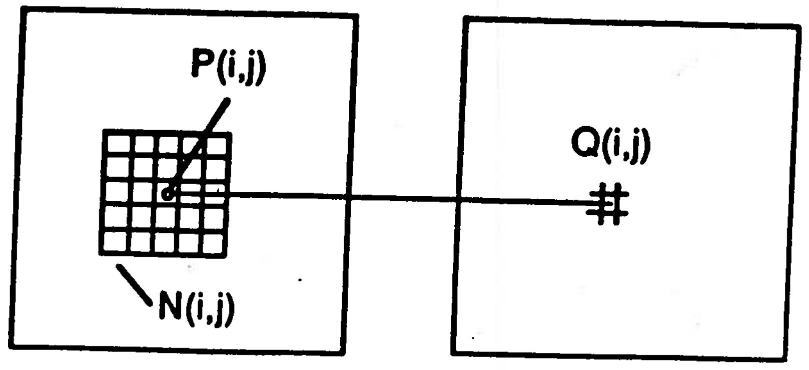


FIGURE 4 Convolving with the FIR function.

$$\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

$$\Delta_x \qquad\qquad \Delta_y$$

FIGURE 5 The Sobel edge detector kernels.

A frequently used example of an algorithm involving the convolution of the image with FIR filters is the use of the Sobel operators for edge detection[4]. The image is convolved with each of the two FIR kernels $\Delta_x$ and $\Delta_y$ shown in Figure. 5. The result of the convolution are the two images $e_x$ and $e_y$ where:

$e_x(i,j)$ is an $M \times M$ array of $x$-direction edge (gradient) strengths.
$e_y(i,j)$ is an $M \times M$ array of $y$-direction edge (gradient) strengths.

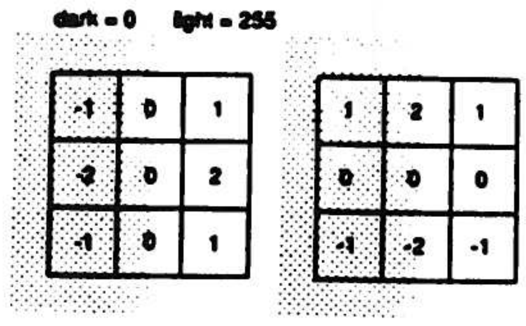
These two images are then combined to form a combined edge strength array, $E$, and an edge direction array, $\theta$, where

$$E(i,j) = \sqrt{e_x^2 + e_y^2},$$

and

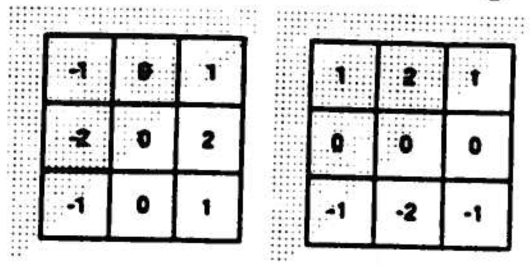
$$\theta(i,j) = \arctan\left(\frac{e_y}{e_x}\right) - \frac{\pi}{2}.$$

We assume that the bright side is to the left of the edge when pointing in the direction of the edge. A numerical example that illustrates the Sobel edge detection operators is shown in Figure 6. The heap-of-parts image is shown in Figure 7 after the application of the Sobel operators.



$$e_x = \frac{1020}{4} = 255 \qquad e_y = 0$$

$$E = 255 \qquad \Theta = \arctan(0) - \frac{\Pi}{2} = -\frac{\Pi}{2}$$



$$e_x = 191.25 \qquad e_y = -191.25$$

$$E = 270.47 \qquad \Theta = \arctan\left(\frac{-1}{1}\right) - \frac{\Pi}{2} = -\frac{3\Pi}{4}$$

**(E Would be 255 if $\frac{1}{\sqrt{2}}$, 1, $\frac{1}{\sqrt{2}}$ weights were used.)**

FIGURE 6 A numerical example of the Sobel operators.

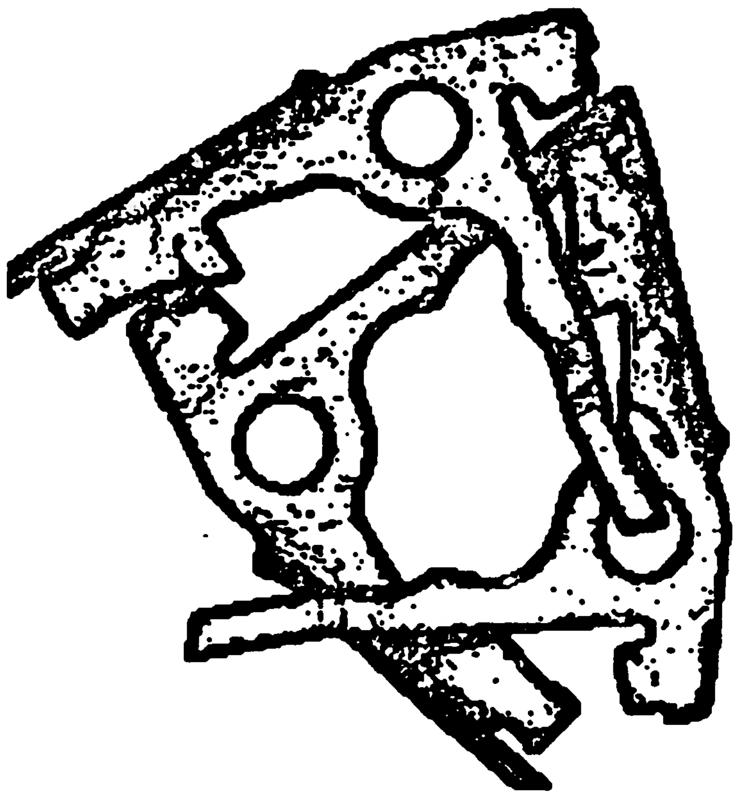


FIGURE 7 The heap-of-parts image after the Sobel operators.

## *Convolve or Transform*

When the convolution kernel $K(i,j)$ becomes large enough, it is more computationally efficient to perform the convolution in the frequency domain by utilizing the convolution theorem[5,23]:

$$K * P = \mathcal{F}^{-1}(\mathcal{F}(K) \times \mathcal{F}(P)).$$

Where K and P are the kernel and the image to be convolved respectively, * denotes the convolution operation, $\mathcal{F}$ is the Discrete Fourier Transform (DFT) and $\mathcal{F}^{-1}$ is the inverse DFT.

In the spatial domain, the time to convolve the $M \times M$ image $P$ with the $m \times m$ kernel $K$, in units of the time for an addition, is given by

$$T_s = m^2M^2 \ (l + \mu),$$

where $\mu$ is the time for a multiplication in units of the time for an addition (we have ignored boundary effects).

In the frequency domain, it is necessary to extend both the image and the kernel with 0's so they have the same period $L$, i.e., they are both $L \times L$ matrices. To avoid wraparound errors, the value of $L$ must be chosen such that the condition $L \geq M + m - 2$ is satisfied[5].

The DFT of an $L \times L$ matrix $Q$, which is given by

$$\mathcal{F}(P(k,l)) = \sum_{i=0}^{L-1}\sum_{j=0}^{L-1} P(i,j)W_L^{ki}W_L^{lj},$$

where $W_L = exp\left(\frac{-2\pi\sqrt{-1}}{L}\right)$, can be split into two Fast Fourier Transforms (FFT's) and performed in $L^2\log_2(L)$ complex multiplications and $2L^2\log_2(L)$ complex additions. The same count of operations is incurred when the inverse DFT is performed.

Therefore, the application of the convolution theorem can be performed in $3L^2\log_2(L) + L^2$ complex multiplications and $6L^2\log_2(L)$ complex additions. Assuming that a complex addition is two normal additions and a complex multiplication is four normal multiplications plus two normal additions, then, in units of time for an addition, the time to convolve the image in the frequency domain is given by

$$T_f = 6(3 + 2\mu)L^2\log_2(L) + 2(1 + 2\mu)L^2.$$

Then direct convolution should be used only if

$$T_s < T_f.$$

Assuming $\mu \gg 1$ and $M \gg m$ then the above condition becomes

$$m^2 > 12\log_2(M) + 4,$$

which requires that $m$ be about 9 for a $512 \times 512$ image.

The above results hold only for the serial implementation of convolution. In the case of parallel implementation, data movement time, if necessary, must be taken into consideration as well.

## *Non-Maximal Suppression*

The non-maximal suppression algorithm is used to thin the thick edges obtained by edge operators such as the Sobel edge operator to exactly one pixel wide edges. It performs that function by suppressing (i.e. eliminating) locally non-maximal strength edge points in the direction normal to the edge direction[6].

The algorithm is applied to each pixel $P(i,j)$ in the input image and needs edge strengths at 3 points in the image: the pixel $P(i,j)$ and two points that lie on the normal to the edge direction at the pixel $P(i,j)$. This is depicted in Figure 8. The edge vector at $P(i,j)$ which has magnitude $E(i,j)$, the edge strength, and slope $\theta(i,j)$, the edge direction, is shown as the vector $\vec{E}$. The two points $P_s$ and $P_n$ are obtained by extending the normal to the edge direction

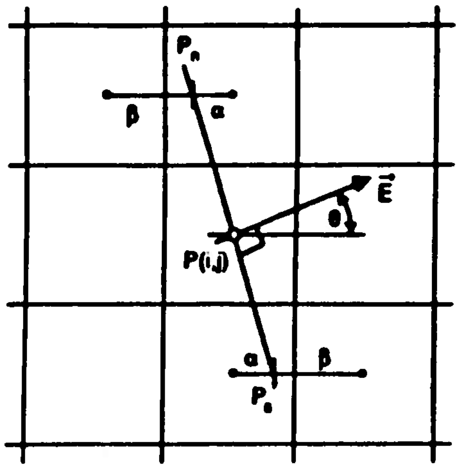


FIGURE 8 Non-maximal suppression.

as shown in the figure. As the edge strengths are available only at discrete intervals, it becomes necessary to interpolate for the edge strengths at points $P_s$ and $P_n$ as follows:

$$E_n = \beta E(i-1,j) + \alpha E(i-1,j-1)$$

and

$$E_s = \beta E(i+1,j+1) + \alpha E(i+1,j).$$

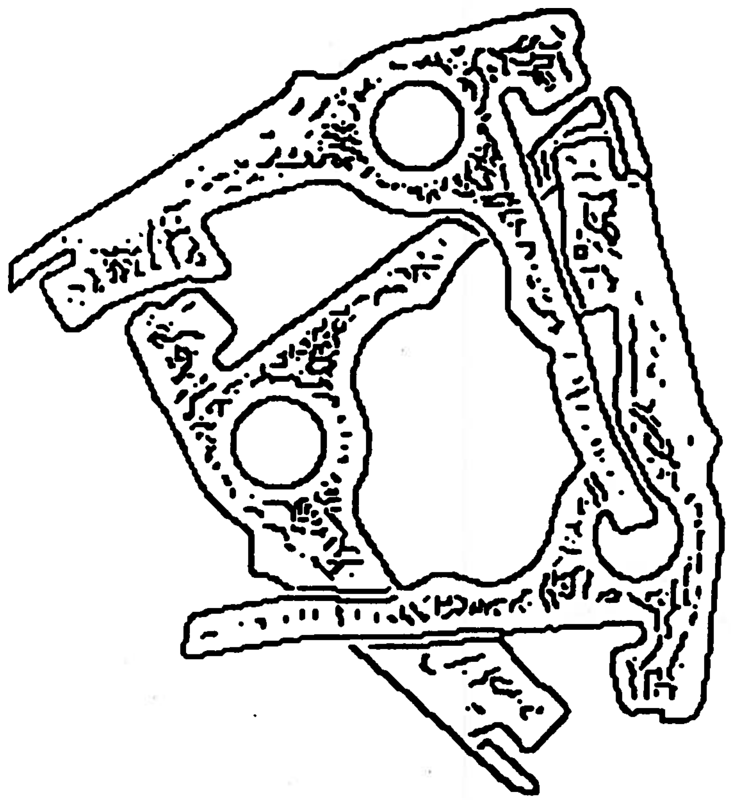


FIGURE 9 The heap-of-parts image after low level operations.

where $E_s$ and $E_n$ are the edge strengths at points $P_s$ and $P_n$ respectively, $\alpha = \left|\frac{e_x}{e_y}\right|$ and $\beta = 1 - \left|\frac{e_x}{e_y}\right|$.

The suppression of the non-maximal point is then implemented as

$$\text{If } E(i,j) < E_n \textit{ or } E(i,j) < E_s \text{ then eliminate } E(i,j).$$

The heap-of-parts image is shown in Figure 9 after edge detection and the non-maximal suppression.

## Intermediate Level Processing

The stage of intermediate level processing involves the extraction of features from the enhanced image produced by low level processing. The extracted features are then re-represented in a more convenient data structure to facilitate their high level processing. Hence, this stage is basically a transducer stage between the low level and the high level processing stages. This stage of processing has the following general characteristics:

1. The representation of the image in the input to the processing is a low level representation.
2. The representation of the image in the output is a set of attributes that explicitly describe the features in the image. That is, the low level representation of the image at the input, in which the features in the image are not explicit, is transformed into another representation in which features are explicitly and, hence, more conveniently represented. In general, it is difficult to characterize the data structure representation at the output of intermediate level processing any further. This can be attributed to the fact that efficient and convenient representations of different features require different data structures. Furthermore, different algorithms may elect to represent the same feature in different ways depending on the use of the feature in the algorithm. However, the following can be said about these representations:
   (a) The spatial information regarding the location of the pixels is explicit in the representation.

   (b) The representation of a pixel value is generally restricted to a small set of values indicating the membership of the pixel in a feature set.
3. The set of operations applied to the pixels of the image generally depends on the value of the pixel and/or its context. As a consequence, equal size areas of the image take different amounts of processing times.

The following algorithms form the benchmark for intermediate level algorithms.

### *Edge Following Algorithm*

Although edge detection and non-maximal suppression yield one pixel wide edges, these edges are still not related or organized in any way that facilitates their further processing. Furthermore, there is a large number of pixels that do not represent any edge points and, hence, are of no further interest. The edge following algorithm can be used to extract the edge points and organize them in the form of connected boundary segments. It also discards pixels that are not part of these boundary segments.

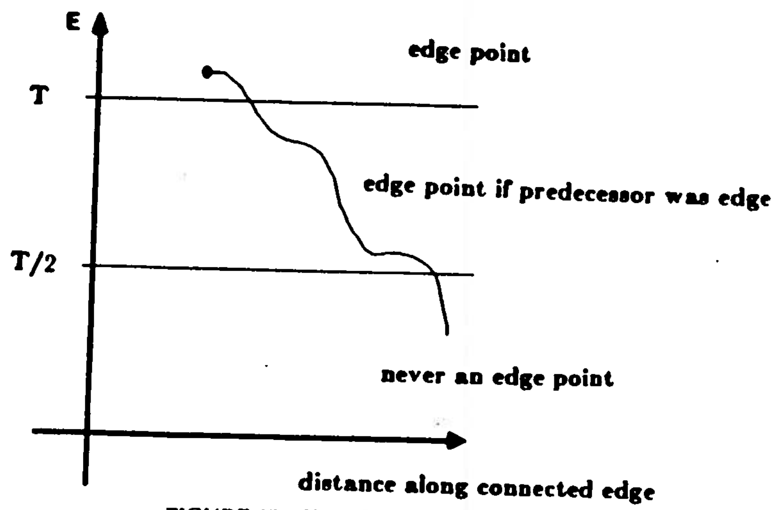

E
T
T/2
edge point
edge point if predecessor was edge
never an edge point
distance along connected edge

FIGURE 10 The edge following algorithm.

The edge following algorithm works as follows[7]. Starting at any edge pixel in the image, the algorithm traces the boundary by moving from one edge pixel to its neighboring one. If at any pixel the edge strength is greater than $T$, where $T$ is a predefined threshold, then the pixel belongs to the boundary segment. If the edge strength at the pixel falls between $\frac{T}{2}$ and $T$, then the pixel belongs to the boundary only if its predecessor pixel is already on the boundary. Finally, if the edge strength at the pixel is less than $\frac{T}{2}$ then that pixel is discarded. At this point the boundary segment terminates and all edge pixels that belong this boundary segment are removed from the image and are represented by a linked list. Each link in the list has information regarding the edge strength, edge

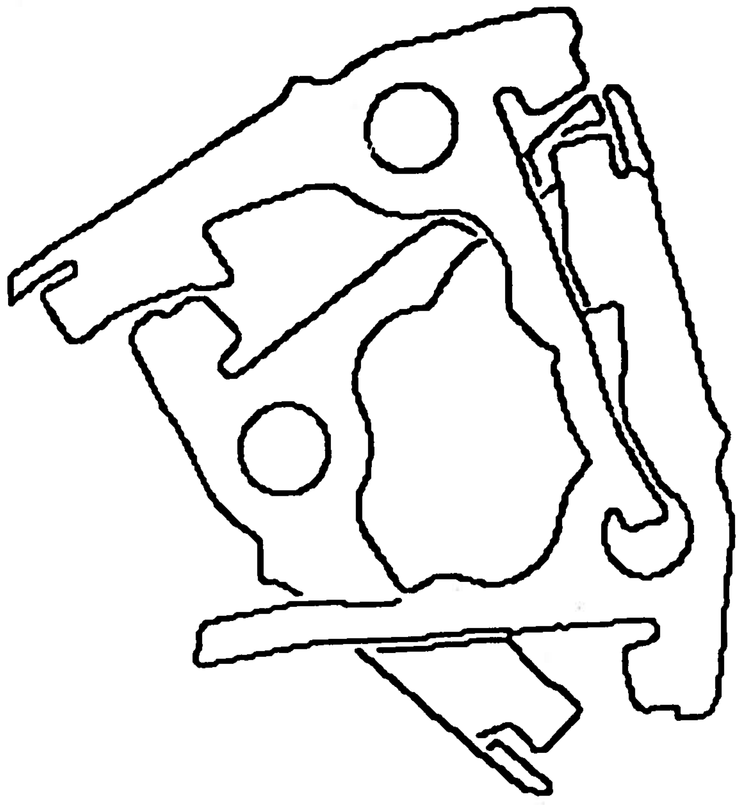


FIGURE 11 The heap-of-parts image after the edge following algorithm.

direction and coordinates of one pixel in the boundary segment. The edge following process is depicted in Figure 10. The above process then repeats starting at a new edge point in the image. The edge following algorithm terminates when there are no more edge points in the image. The output of the algorithm is, hence, a set of linked lists each describing a boundary segment in the image.

The edge following algorithm illustrates the basic characteristics of intermediate level algorithms. The features, in this case edge pixels, are extracted from the image and re-represented along with explicit address information in another data structure, in this case a linked list. Furthermore, the processing of equal area images require, in general, different amounts of processing times as the images contain, in general, unequal counts of edge pixels.

The heap-of-parts image is shown in Figure 11 after the edge following algorithm.

## *The Hough Transform*

The Hough transform algorithm [7,25] can be used to identify edge pixels that form straight lines in the image. It does so by considering all possible lines at once and then rating each on how well it explains the the original image data.

The basic concept behind the Hough transform is the relation between points in the *image space* and lines in the *parameter space*. This relation is illustrated in Figure 12. For a point $(X_1, Y_1)$ in the image space, the set of all possible lines that can pass through that point must satisfy the equation $Y_1 = mX_1 + c$. If $(X_1, Y_1)$ are considered to be fixed, allowing $m, c$ to be variables, then this point corresponds to the line $c = -X_1m + Y_1$ in the $c$-$m$ space or the parameter space. Hence, a point $(X,Y)$ in the image space corresponds to a line in the parameter space governed by the equation $Y = mX + c$. Then if the two points $(X_1,Y_1)$ and $(X_2,Y_2)$ are connected by the line $AB$, which is described by the equation $Y = m_1X + c_1$, then there must exist two lines in the parameter space corresponding to these two points which intersect in the point $(m_1,c_1)$. By using the same argument, all points that are on the line $AB$ must correspond to lines in the parameter space that intersect at the point $(m_1,c_1)$.

The relation between the image space and the parameter space can be used to implement the Hough transform as follows. The

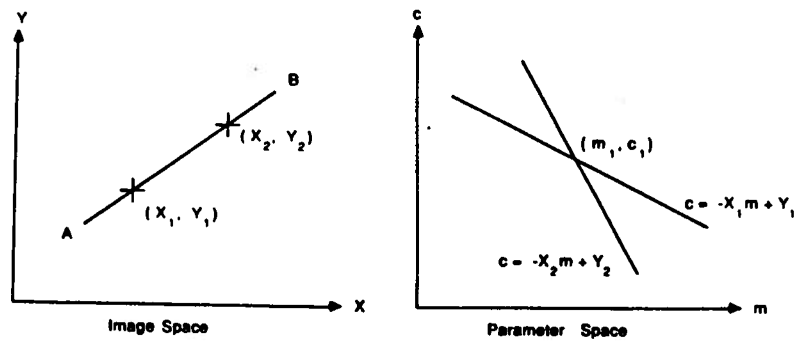


FIGURE 12 The relation between image space and parameter space.

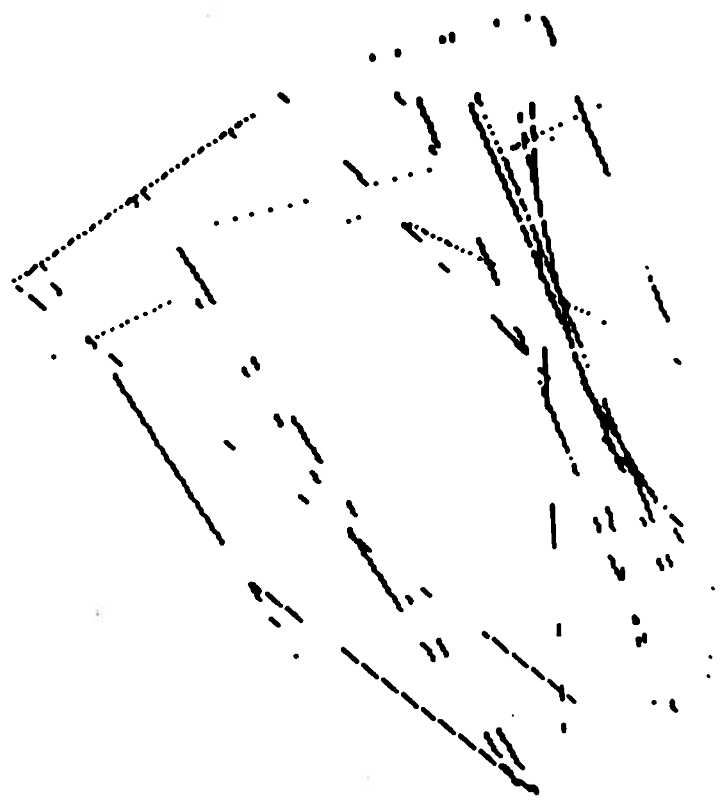


FIGURE 13 The heap-of-parts image after the Hough transform.

parameter space is quantized in the form of an array that is referred to as the accumulater array and is initialized to zeros. Then for each edge point in the image array, all accumulator array elements that are intersected by the line corresponding to the edge point are incremented by 1. Edge points that lie on straight lines in the image array intersect at the same location in the accumulater array and, hence, cause maximum accumulation. Therefore, if local maxima in the accumulater array are detected, the parameters of all straight lines in the image can be obtained. The values of the accumulater array elements give a measure of the number of points in each line.

The heap-of-parts image is shown in Figure 13 after the Hough transform.

## High Level Processing

The objective of high level processing is generally the recognition of objects in the image. This is done through analysis, classification and identification of features extracted from the image during earlier low and intermediate level processing. Hence, high level processing may be generally characterized by analysis of feature lists that aim at the identification of objects in the image. The high level processing stage is basically the stage of image understanding.

High level processing and high level algorithms are, in general, less well understood and more difficult to characterize than low level and intermediate level processing and algorithms. This is basically attributed to:

1. High level algorithms generally involve symbolic processing of feature lists and/or employ techniques from several areas such as calculus, graph theory, differential geometry, category theory, logic and artificial intelligence. These techniques posses very diverse characteristics.
2. There is no uniform structure for the representation of the image in high level processing. In fact, high level algorithms vary considerably in this aspect. This is contrary to low level algorithms, for example, in which the representation of the image data is very uniform (two dimensional arrays of pixels).

We have, somewhat arbitrarily, chosen the following algorithms for our benchmark.

### Graph Theoretic Algorithms

Graph theoretic algorithms refer to a class of high level processing algorithms that classify and recognize objects in the image by examining features extracted from the image against pre-defined models of the features of possible objects that can appear in the image[7]. A pre-defined model of the features of an object is referred to as the *template* of that object. That is, graph theoretic algorithms are matching algorithms in the general sense.

The features extracted from the image are represented as a directed graph $(V,E)$ which consists of a set of vertices $V$ and a set of edges $E$. The vertices represent the features extracted from the image. The edges represent relationships among these features. The template of an object is also represented in the same form. This representation of features is referred to as a *relational structure*[7].

Hence, the process for matching an object to a template can be formalized as one of the graph isomorphism related problems, such as graph isomorphism, subgraph isomorphism and double subgraph isomorphism.

In graph isomorphism, given the image data graph $(V_1, E_1)$ and the template graph $(V_2,E_2)$, the objective is to find a one-to-one onto mapping (i.e. an isomorphism) $f$ between $V_1$ and $V_2$ such that for $v_1, v_2 \in V_1, V_2$ $f(v_1) = v_2$ and for each edge $E_1$ connecting any pair of vertices $v_1, v_2 \in V_1$, there is an edge $E_2$ connecting $f(v_1)$ and $f(v_2)$. In subgraph isomorphism, the objective is to find isomorphisms between the template graph and subgraphs of the image data graph. In double subgraph isomorphism, the objective is to find *all* isomorphisms between subgraphs of the template and subgraphs of the image data graph.

Graph isomorphism related techniques as described above are pure matching techniques. That is, the match between the template and the object should be perfect otherwise no-match is obtained. This can limit the usefulness of these techniques in the case of noisy input and/or imprecise templates. It is possible, however, in the implementation of the technique, to relax the strict rules of isomorphism correspondence and obtain a computational version of the technique that can take into consideration noise and imprecise templates. This is generally done by introducing matching metrics that measure the goodness of the match and hence, provide a quantified measure of matching. There are several methods for implementing

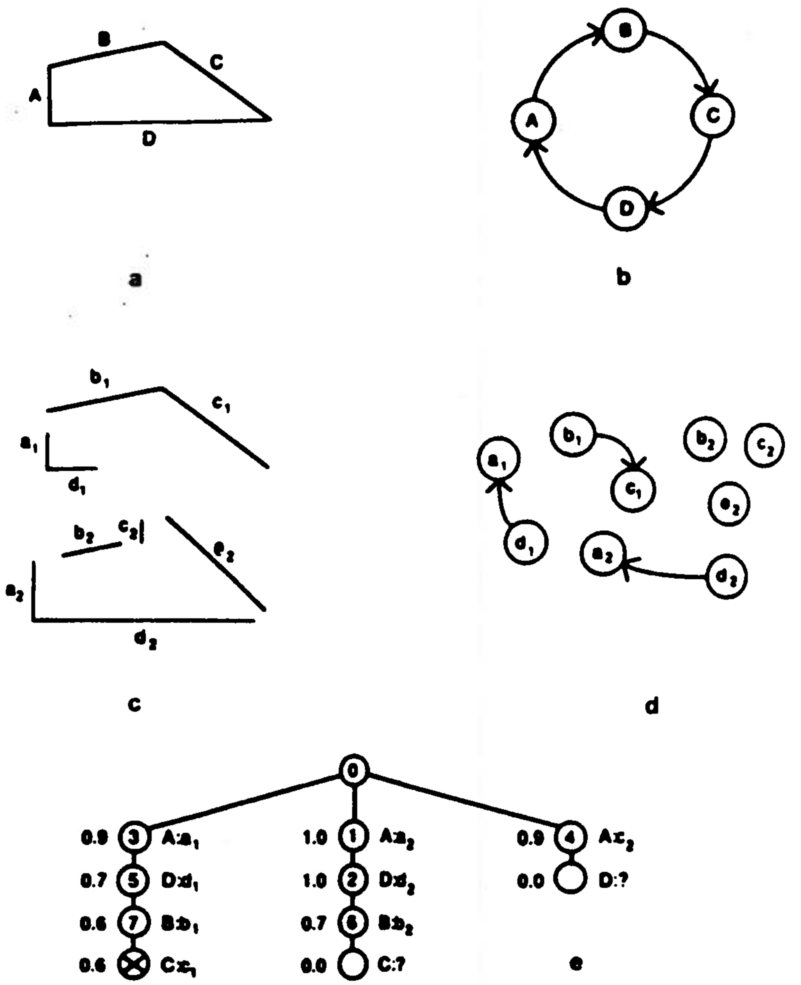


FIGURE 14 An example illustrating backtrack search.

graph isomorphism techniques with matching metrics. However, only the technique referred to as *backtrack search* is considered in this chapter.

The backtrack search technique refers to a set of possibly exhaustive search algorithms such as depth-first search, breadth-first search and best-first search. These algorithms differ in the order by which the search process proceeds. Therefore, only best-first search is described.

A generic version of the best-first backtrack search technique is illustrated using the simple example in Figure 14. The straight line segments extracted from the image and the relation 'next to' are used to construct the directed graph for the image data. This is shown in Figure 14(c), (d). The template of the object to be located and its pre-computed directed graph are shown in Figure 14(a), (b) respectively.

The backtrack search locates the object in the image in steps by matching a line segment in the template to a line segment in the image in each step until all line segments in the template are matched. A heuristic function is used to aid the search after each step by returning a value that indicates the goodness of the matches possible from that step. The heuristic function takes into consideration several factors such as how well the two straight lines match and how important the particular line segment in the template is in distinguishing the object in question from other objects that can appear in the image. This is referred to as the *saliency* of the line segment[8]. The search continues in the direction of the largest value returned by the heuristic function up to that point in the search.

The resultant search tree is shown in Figure 14(e). The letters to the right of each node in the tree indicate the two line segments matched at this node. The number to the left of each node is the value returned by the heuristic function. The integer inside each node indicates the order in which the nodes were expanded (decreasing heuristic function value). The search terminates when the object is located after matching $C$ to $c_1$.

### *The Consistent Labeling Problem*

The consistent labeling problem refers to the problem of assigning labels consistently to objects in the image. This problem is also known as relaxation labeling and constraint satisfaction[9]. Consistent labeling algorithms are generally employed after all objects in the image are identified.

The consistent labeling problem has four basic components: a set of objects identified in the image, a finite set of relations among the

objects, a finite set of labels and a set of constraints that determine which labels may be assigned to which objects. Hence, the problem may be stated as: given the input as a relation structure (the objects and their relations), the labels and the constraints, assign labels to objects without violating the constraints.

The solution to the consistent labeling problem can be implemented in a number of ways[7]. It is also possible, in general, to employ backtrack search techniques to solve the consistent labeling problem[10,11].

## ARCHITECTURES FOR ROBOT VISION

Research in special purpose architectures for Robot Vision has progressed considerably over the past decade. This has resulted in a large number of special purpose architectures for Robot Vision. With recent advances in computer hardware technology which is causing a steady decline in computer hardware cost, it is becoming feasible to build such special purpose architectures.

Special purpose architectures for Robot Vision can be divided

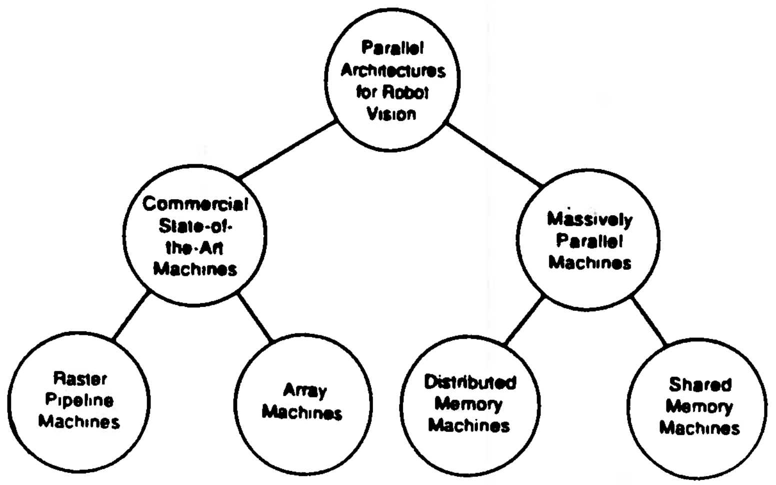

Parallel Architectures for Robot Vision
Commercial State-of-the-Art Machines
Massively Parallel Machines
Raster Pipeline Machines
Array Machines
Distributed Memory Machines
Shared Memory Machines

FIGURE 15 Taxonomy of special purpose architectures for Robot Vision.

into two major groups: *Commercial state-of-the-art* machines, and *Massively parallel* machines. The first group of architectures can be further subdivided into *Raster Pipeline* machines and *Array* machines. The second group of architectures may also be subdivided into *Distributed Memory* machines and *Shared Memory* machines. This taxonomy of special purpose architectures for Robot Vision is shown in Figure 15.

In the following sections, the basic features of each group of architectures are considered. The performance of each group on the set of benchmarks is also discussed.

## Commercial State-of-the-Art Machines

The commercial state-of-the-art machines group of architectures refers to special purpose architectures that are currently available commercially. These machines are generally characterized by:

1. Low cost: the use of state of the art technology makes it possible to build these machines at low cost.
2. Most commercial state-of-the-art machines employ fixed point arithmetic. However, with advances in computer hardware technology, floating point arithmetic is starting to appear.
3. Machines that belong to this group of architectures function basically as attached processors to a host computer. The host computer, usually a conventional one, supervises the operation of the attached processor and performs functions such as loading and unloading of data and programs. The host computer also performs most I/O operations.
4. These machines are intended for Robot Vision applications which are generally simple in nature such as simple inspection of factory samples and aiding CAD graphics computations. There is also a reasonable amount of software available that implement these applications.

### *Raster Pipeline Machines*

The basic architecture of Raster Pipeline machines is depicted in Figure 16. It employs a number of processing elements that are

cascaded in series to form a pipeline of processing stages. The processing stages are controlled by the controller which issues instructions to the stages through a stage instruction bus during a setup phase and then streams the image data into the pipeline during the processing phase. The image buffer is used to hold the image data to be streamed into the pipeline as well as receive the output data from the pipeline. The host computer supervises the overall operation of the pipeline, the controller and the image buffer. Hence, the processing stages, the image buffer and the controller all function as an attached processor to the host computer [12].

Images can enter the pipeline in two ways: (1) the host sends the image to the image buffer where the controller streams it through the pipeline, back into the buffer and finally back to the host; and (2) the host sends and receives and image to and from the pipeline directly.

Images enter the pipeline as a steam of pixels in a sequential line-scanned raster format and move through the pipeline at a constant rate. Shift registers within each stage store two contiguous scan lines of the image. Furthermore, window registers, also within each stage, hold the pixels that form a 3 × 3 neighborhood. Each stage performs a pre-programmed function on this neighborhood. This function is referred to as the neighborhood function. The neighborhood function involves the transformation of the pixel in the center of the neighborhood based on that pixel value plus the values of its eight neighbors. The line and window registers are depicted in Figure 17.

At each discrete time step, a new pixel is clocked into each stage.

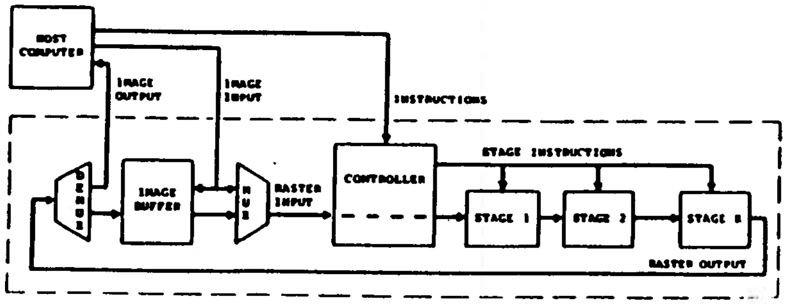


FIGURE 16 Raster Pipeline machines architecture.

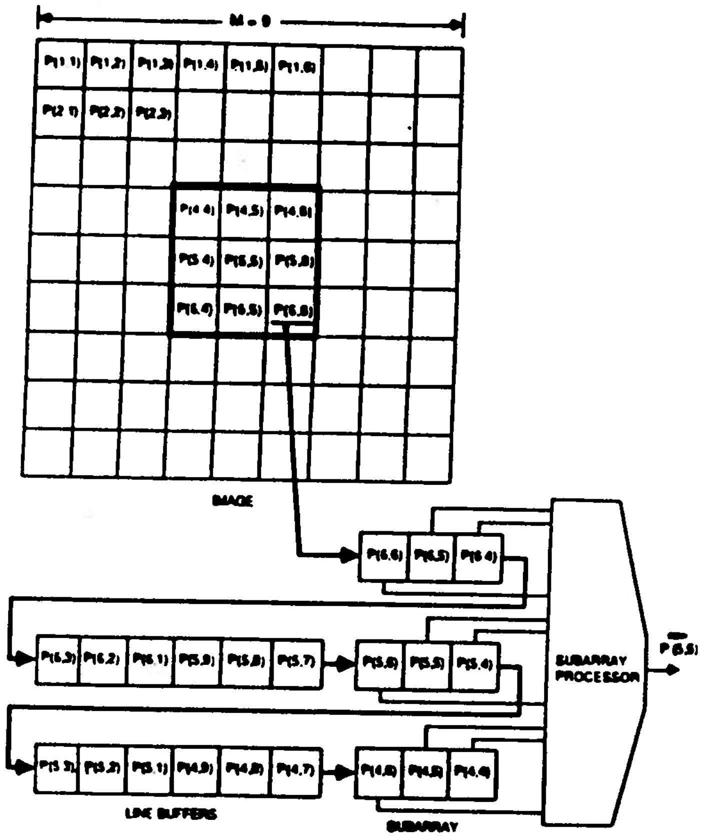


FIGURE 17 Line and window registers.

Simultaneously, the contents of all shift registers are shifted one element. Each stage performs the neighborhood function on the neighborhood it contains obtaining a new value for the center pixel. At the next time step, a new pixel is shifted into each stage and the transformed center pixel is shifted into the next stage. Hence, at each time step, each stage holds a new neighborhood in its window registers. The neighborhoods in three successive stages of the pipeline are shown in Figure 18. The output of the last stage forms the output stream of the pipeline and is fed back to the image buffer or directly to the host.

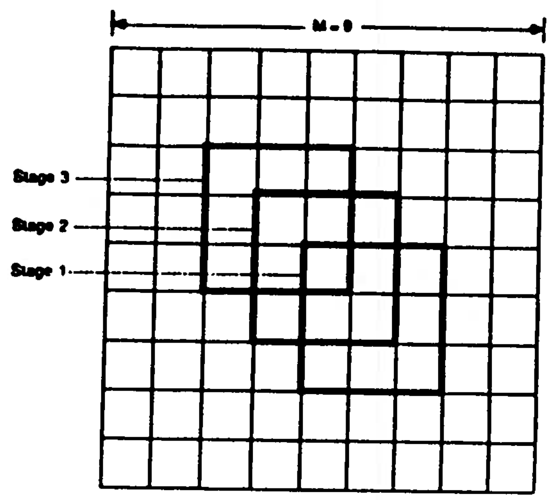


FIGURE 18 Neighborhoods in three successive stages of the pipeline.

Raster Pipeline processors offer a number of advantages:

1. Simple interconnections between the stages. This simplifies design and enhances reliability.
2. The input is in raster scan format which matches the output of many sensors. Hence, the image may be fed directly without reformatting for real-time processing.
3. At steady state, the output is obtained from the pipeline at a constant rate equal to that of the input to the pipeline.

However, there are several disadvantages for Raster Pipeline machines:

1. It is difficult to perform branching. The pipeline has to be flushed first.
2. It is difficult to process multiple images at the same time.
3. It is not possible to handle $1 \times m$ and $m \times 1$ neighborhoods that occur in FFT's and separable kernels.
4. They have a restricted set of instructions.

The cytocomputer is a typical example of Raster Pipeline machines[12,13].

### *Performance on Benchmarks*

*Low Level Processing*. Assuming 16-bit fixed point arithmetic with 100 nsec multiply/accumulate time, Table 2 shows the times needed for convolving the image with an $m \times m$ FIR kernel. Each stage needs $m - 1$ line registers and $m^2$ window registers in order to be able to perform the convolution.

In order to perform the non-maximal suppression algorithm, the images containing the edge strength and the edge direction have to be streamed into the pipeline together, with a resolution of 8 bits per pixel. The non-maximal suppression algorithm involves less computations than a $3 \times 3$ convolution algorithm. Hence, the values for the $3 \times 3$ convolution algorithm in Table 2 are an upper bound for non-maximal suppression execution time provided additional hardware is available in each stage to implement the decision making required in this algorithm.

*Intermediate Level Processing*. The image array must be streamed through each stage of the pipeline $K$ times, where $K$ is proportional to the length of the longest sequence of connected pixels having strengths in the range $(T, \frac{T}{2}]$ before a strength $T$ pixel occurs. That is, $K \propto \frac{1}{m}$, where $m^2$ is the number of window registers in the stage.

It is virtually impossible to implement the Hough transform algorithm on Raster Pipeline machines.

*High Level Processing*. The operations of high level processing algorithms are all virtually impossible to implement on a Raster Pipeline machine. Implementing search techniques, involved in high level algorithms, requires the capability of dealing with dynamic data structures. The design and instruction set of these architectures can not provide this capability.

### *Array Machines*

The general block diagram of the Array machines architecture is depicted in Figure 19. It consists of an array processor (AP) which has its own local memory. The AP is generally a high speed processor that is optimized for vector and array operations. The AP communicates to the host computer through a high speed DMA channel. The image array is sent to the AP memory by the host over

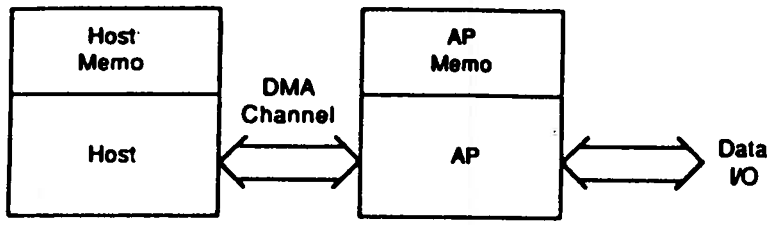


FIGURE 19 The architecture of Array machines.

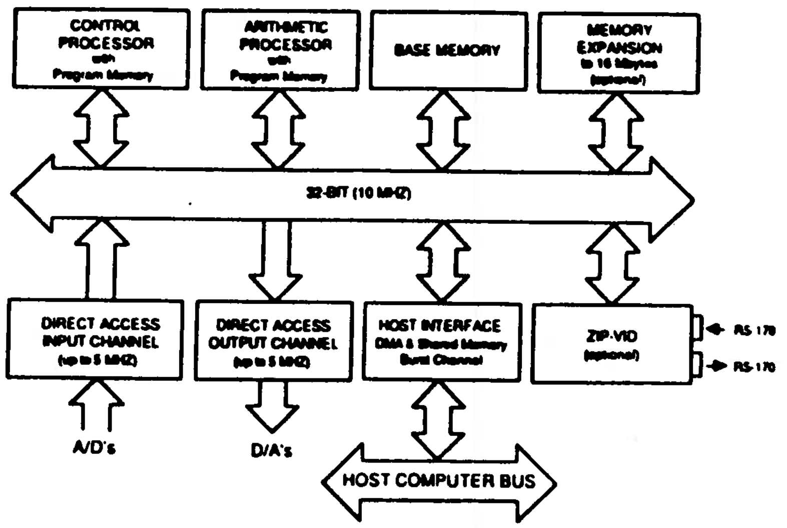

CONTROL PROCESSOR with Program Memory
ARITHMETIC PROCESSOR with Program Memory
BASE MEMORY
MEMORY EXPANSION to 16 Mbytes (optional)
32-BIT (10 MHZ)
DIRECT ACCESS INPUT CHANNEL (up to 5 MHZ)
DIRECT ACCESS OUTPUT CHANNEL (up to 5 MHZ)
HOST INTERFACE DMA & Shared Memory Burst Channel
ZIP-VID (optional)
RS 170
RS-170
A/D's
D/A's
HOST COMPUTER BUS

FIGURE 20 The ZIP3216 array processor.

the DMA channel. The AP then performs the required computation on the image array at high speed. The result is then sent back to the host over the DMA channel. Therefore, the use of the term *array* in this context reflects the nature of the data being processed.

Array machines offer several advantages:

1. Array machines offer more flexible programming than Raster Pipeline machines.
2. Array machines can easily handle large, symmetric and separable kernels.

3. The Array machine can operate in parallel with the host computer.

A typical example of the Array machines architecture is the ZIP3216 array processor[14]. The block diagram of the ZIP3216 is shown in Figure 20.

*Performance on Benchmarks*

*Low Level Processing*. A typical array processor, the ZIP3216, employs 16 bit fixed point arithmetic with 100 nsec multiply/accumulate time. Table 2 again shows the time for convolution for different kernel sizes.

| | M = 256 | M = 512 |
|---|---|---|
| m | time (sec.) | time (sec.) |
| 3 | 0.06 | 0.24 |
| 5 | 0.16 | 0.66 |
| 7 | 0.32 | 1.28 |
| 9 | 0.53 | 2.12 |
| 11 | 0.79 | 3.17 |

Table 2 Convolution times on Raster Pipeline machines

*Intermediate Level Processing*. The architecture of Array machines allows the processor to ignore non-edge pixels that are not needed for processing. In our experiments, the probability of a pixel being on an edge was typically in the range of 0.1 – 0.2. That is, less than 20% of the image has actually to be processed (20K to 50K bytes in a 256K byte image). Hence, for Array processors, the times for intermediate level processing is less than those for low level processing. This contrasts favorably with Raster Pipeline machines.
*High Level Processing*. Current Array machines do not provide the desired functionality for dealing with dynamic data structures needed for high level processing. In principal they could; however, this type of processing is intended to be done in the host and can be done concurrently with other processing.

## Massively Parallel Machines

The Massively Parallel machines group of architectures refers to parallel architectures that employ a very large number of processors to achieve massive parallelism. The processors are interconnected in some manner to allow the sharing of data. Massively Parallel machines are generally characterized by:

1. High cost: due to the large number of processors and their interconnections, the cost of these machines is generally very high.
2. Massively Parallel machines are not widely available. They typically exist in research labs. In fact, many of these machines have yet to be implemented.
3. Major research must still be conducted in the areas of software and algorithm design for these machines.
4. I/O remains a bottleneck in these machines yet it has received very little attention.

The group of Massively Parallel machines is divided into two major subgroups: *Distributed Memory* machines and *Shared Memory* machines.

### *Distributed Memory Machines*

In this group of architectures, each processor has its own local memory and the processors are interconnected to each other via communication links. The processors use these communication links to share data. The following are some examples of Distributed Memory machines:

*Arrays*

In this case, the processors are interconnected together to form a grid of processors. This is shown in Figure 21(a). Each processor is connected only to its four neighbors in the grid. The term *array*, in this context, is different from its use in the case of commercial state-

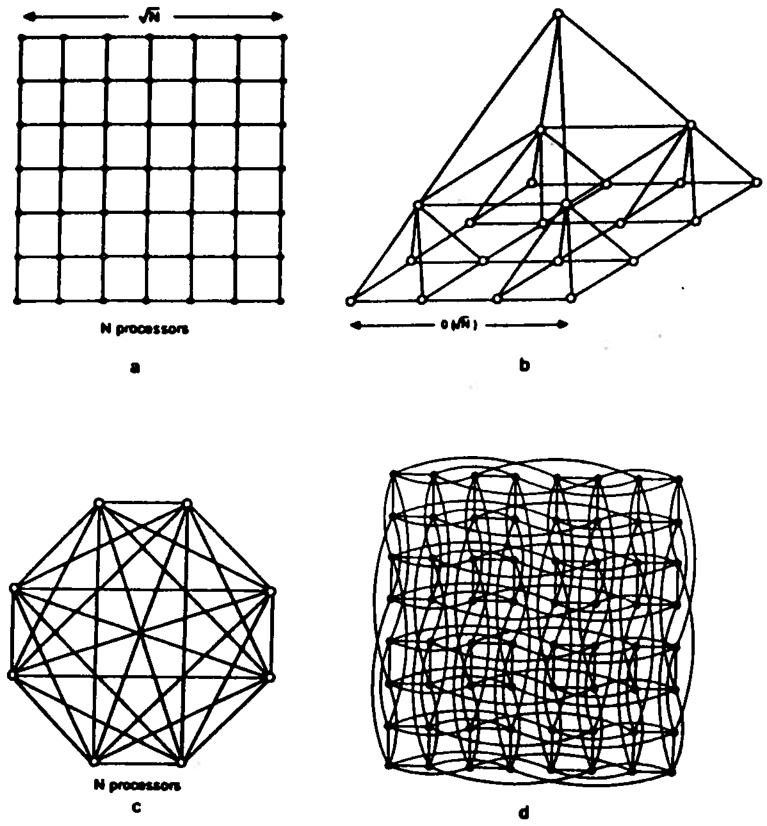


FIGURE 21 Distributed Memory architectures.

of-the-art machines and reflects the two dimensional spatial array of processors that form the grid. The advantage of this architecture is that the interconnection patterns are simple and are independent of the number of processors of the array. The disadvantage is that data movements across the grid beyond the four neighbors are time consuming ($O(\sqrt{N})$).

Examples of the array massively parallel architecture include the Illiac IV[15], which consists of 256 processors organized as four $8 \times 8$ grids of processors; and the Massively Parallel Processor MPP [16], which consists of a $128 \times 128$ grid of bit serial processors.

### *Pyramids*

In this case, the processors are interconnected together to form a pyramid of processors. This is shown in Figure 21(b). Each processor is connected to four neighbors, a parent and four children processors. The advantage of this architecture is the simple interconnection pattern that is independent of the number of processors as well as the short distance between the processors; no two processors are more than $O(\log_2(N))$ steps apart. The disadvantage is congestion in the upper levels during system wide data transfers.

Examples of the pyramid massively parallel architecture include the PMPP[17] and the Pyramid Machine[26].

### *Completely Connected*

In this case, every processor is connected to every other processor in the architecture. This is shown in Figure 21(c). Hence, each processor is exactly one step apart from any other processor. The advantage of this architecture is the high connectivity of the processors. The major disadvantage is the number of interconnections that grows quadratically with the number of the processors.

### *Hypercubes*

In this case, the processors are connected to each other in such a way to form a hypercube array. This is shown in Figure 21(d) for 64 processors. The hypercube architecture offers several advantages:

1. The hypercube architecture is recursive. This makes the architecture recursively partitionable into smaller hypercubes.
2. It is possible to map the hypercube interconnection pattern into other important interconnection patterns such as grids and pyramids.
3. The hypercube architecture provides a good compromise between the decreasing proximity of processors and the number of interprocessor connections.

Examples of the hypercube architecture include the Intel iPSC, the NCUBE and the Ametek hypercubes. Table 3 summarizes and compares their basic features.

| | Intel iPSC | Ametek 14/n | NCUBE/ten |
|---|---|---|---|
| No. nodes (max) | 128 | 256 | 1024 |
| Memory per node | 512KB | 1MB | 128KB |
| CPU | 286/287 16 bit | 286/287 16 bit | Custom 32bit |
| MIPS/node | 0.30 | 0.30 | 2.00 |
| MFLOPS/node | 0.07 | 0.07 | 0.50 |
| Node-to-Node | 150KB/sec | N/A | 1MB/sec |
| I/O Bandwidth | 1MB/sec | N/A | 90MB/sec |

Table 3 Commercially available hypercubes

## *Shared Memory Machines*

In this group of Massively Parallel architectures, each processor contains a limited size local memory and the processors are connected to a set of globally shared memory modules by an interconnection network. This is depicted in Figure 22. Hence, all processors share one common memory.

The interconnection network provides the means by which any processor may access any memory module as well as the means to resolve any conflicts that may occur on memory modules. There are numerous types of interconnection networks ranging from single shared buses to full crossbar switches[24].

The Shared Memory architecture has the advantage of providing a large shared memory for the processors. This is convenient for conventional block structured languages that imply a shared memory architecture such as Ada. However, as the number of processors increase, the interconnection network dominates the system cost and can degrade its performance.

Examples of interconnection network based machines include: PASM[18], which consists of 1024 processors and a reconfigurable multistage interconnection network; and the IBM RP3[19] which uses 512 processors.

The performance of Massively Parallel machines on the benchmark algorithms is presented below for hypercube architectures. The use of the hypercube architecture as a representative for Massively Parallel architectures is motivated by the several advantages this architecture offers and the fact that a number of hypercube machines are already commercially available. A model for hypercube architectures is first presented. This model is then used to evaluate the performance of hypercube architectures on the benchmark algorithms.

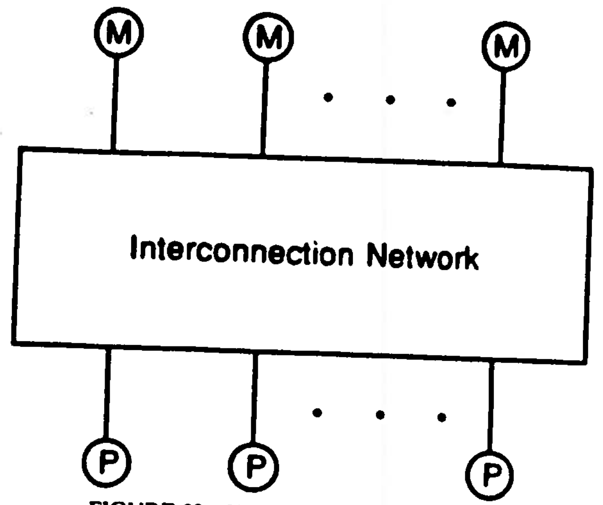


FIGURE 22 Shared Memory architectures.

## *A Model for Hypercube Machines*

An $n$-cube array can be constructed recursively from $N = 2^n$ node processors as follows:

1. Basis Step: Form a 1-cube from 2 processors connected by a single communication link. Label one node 0 and the other 1.
2. General Step: Construct an $n$-cube from two $(n - 1)$-cubes as follows. First prefix the node labels in one of the $(n - 1)$-cubes with an 0. Second, prefix the node labels in the other $(n - 1)$-cube with a 1. Finally connect the two $(n - 1)$-cubes with communication links between pairs of nodes that have labels differing only in their most significant bit. A 4-cube (16 nodes) is shown in Figure 23.

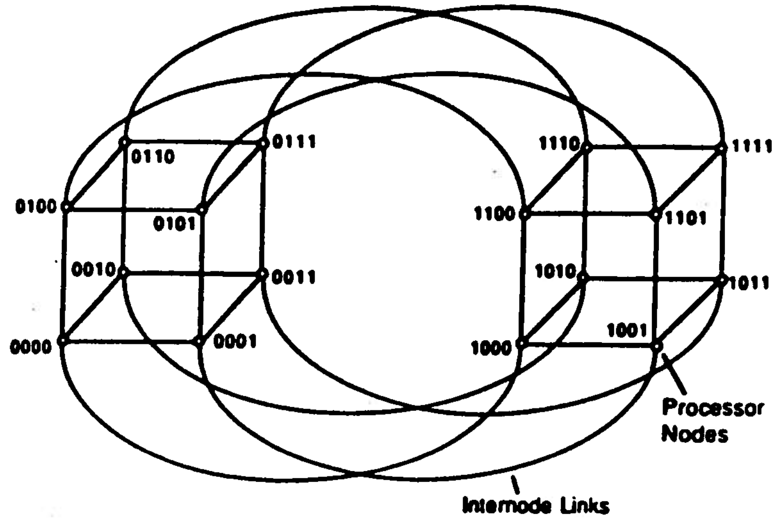


FIGURE 23 A 4-cube.

The cube array is connected via a set of I/O channels to a cube manager. This is shown in Figure 24. The cube manager is used for I/O (disks, tapes, cameras, sensory devices, etc.), as a peripheral controller and for program development. The cube array and the cube manager are referred to as the hypercube system.

A model for the node processor is shown in Figure 25. It consists of a CPU with a cache or large register file, main memory and $n + 1$ bidirectional DMA channels. The first $n$ of the DMA channels are used to join the node processor to its nearest $n$ neighbors in the cube array. The $(n + 1)^{st}$ DMA channel provides a link for communicating with the cube I/O system. It is assumed that caching within a node allows the DMA to proceed so that a fraction $\gamma$ of the internode communication time can be overlapped with the node processing. The factor $\gamma$ is referred to as the *degree of transparency*.

The time for an algorithm to run on a hypercube is given by:

$$T(N) = T_i + T_p(N) + (1 - \gamma)T_c(N) + T_o \qquad (1)$$

where $N$ is the number of node processors in the cube array, $T_p$ is the time of perform the processing, $T_c$ is the internode communication time, $T_i$ is the time to input the image data, $T_o$ is the time to

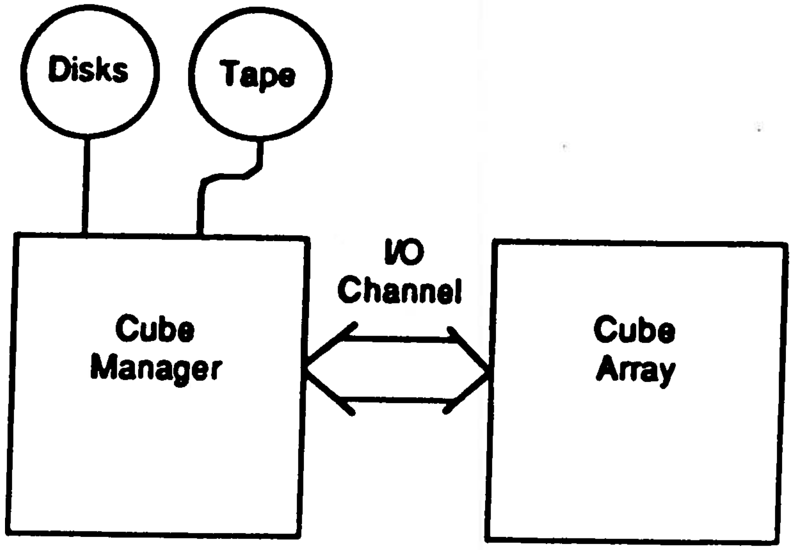


FIGURE 24 The hypercube system.

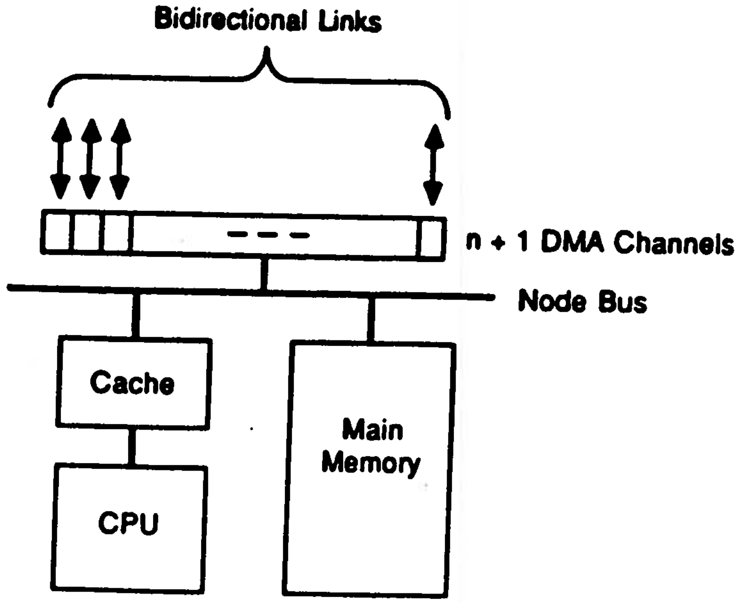


FIGURE 25 The node processor.

output the image data, and $\gamma$ is the degree of transparency. Ignoring the I/O time, equation 1 becomes:

$$T(N) = T_p(N) + (1 - \gamma)T_c(N) \tag{2}$$

There are two principal contributors to intrinsic inefficiency of parallel algorithms on Massively Parallel architectures such as the hypercube. The first is the communication overhead which is incurred when $\gamma < 1$. The other is the dependencies within the algorithm that do not permit all $N$ processors to be used all the time. This is reflected by the node efficiency, given by:

$$E_p(N) = \frac{T_p(1)}{NT_p(N)} \leq 1$$

The overall system efficiency is given by:

$$E(N) = \frac{T(1)}{NT(N)} \tag{3}$$

Using equations (2) and (3), and noting that $T_c(1) = 0$, then:

$$E(N) = \frac{E_p(N)}{1 + (1 - \gamma)\dfrac{T_c(N)}{T_p(N)}}$$

We define a *perfectly scalable* algorithm as one for which $E(N) = 1$, $N > 1$. This requires that $E_p(N) = 1$, i.e., processing is 100% efficient; and $(1 - \gamma)T_c(N) = 0$, i.e., the communication overhead is zero. Loosely speaking, a perfectly scalable algorithm can make use of large number of processors without diminishing returns. In general, it is desirable to design the parallel algorithm to be perfectly scalable[20].

### *Performance on Benchmarks*

#### *Low Level Processing*

The image is partitioned into subimages of equal sizes, each subimage assigned to a node processor. A natural assignment for the hypercube is to partition the $M \times M$ image into a Gray code of $2^{n/2} \times 2^{n/2}$, ($n$ assumed even), subimages similar to an $n$-dimensional Karnaugh map, and then to place each subimage with its like numbered processor. This is shown in Figure 26 for the hypercube shown in Figure 23. This method for partitioning the image and assigning the partitions to node processors guarantees that adjacent subimages are in adjacent processor nodes[20].

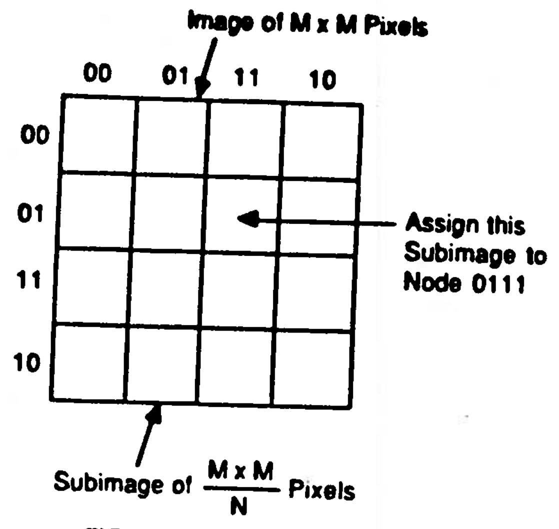


Image of M x M Pixels
00
01
11
10
00
01
11
10
Assign this Subimage to Node 0111
Subimage of $\frac{M \times M}{N}$ Pixels

FIGURE 26 Partitioning the image.

The algorithms in the benchmark for low level algorithms can be implemented as identical programs running in parallel, each in a processor node. That is, the hypercube is simulating an SIMD (Single Instruction stream, Multiple Data stream) machine. Hence, $E_p(N) \approx 1$ and the only potential contributor to inefficiency would be the communication overhead that results from the need to exchange data around the edges of the subimage to implement neighborhood operations. This is depicted in Figure 27. It shows a subimage in node $A$ and the data that has to be moved from adjacent nodes. The number of pixels that has to be transferred is given by:

$$\frac{4M}{\sqrt{N}}\left\lfloor \frac{m}{2} \right\rfloor + (m - 1)^2$$

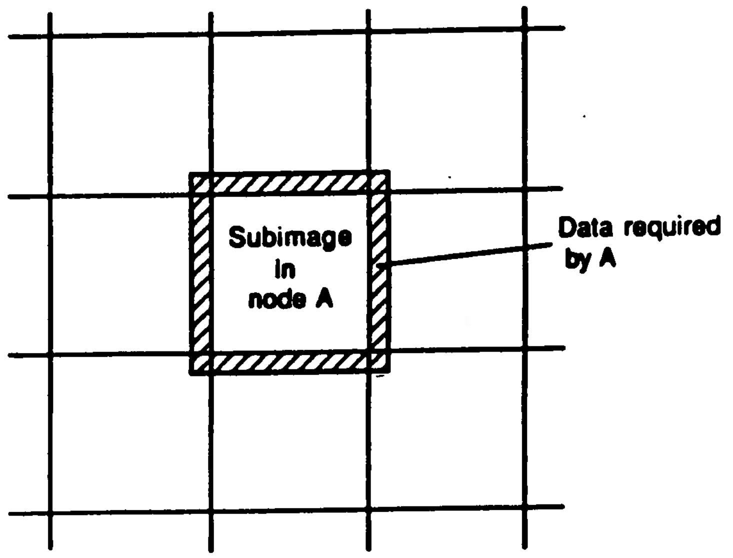


FIGURE 27 Data needed by a subimage in node A.

where $m \times m$ is the size of the kernel used and $M \times M$ is the size of the image. $N$ is the number of node processors. The communication time necessary to move the pixels is proportional to their number. Hence:

$$T_c(N) = K_2\left[\frac{4M}{\sqrt{N}}\left\lfloor\frac{m}{2}\right\rfloor + (m-1)^2\right] \qquad (K_2 \textit{ is a constant}) \qquad (4)$$

The processing time at a node is proportional to the number of pixels in that node. All nodes have the same number of pixels. Hence:

$$T_p(N) = K_1 m^2 \frac{M^2}{N} \qquad (K_1 \textit{ is a constant})$$

To insure perfect scalability, the following is required:

$$E_p(N) = 1 \qquad (\textit{which is true})$$

and

$$(1 - \gamma)T_c(N) \ll T_p(N).$$

That is,

$$\frac{K_2}{K_1}(1 - \gamma)\left[\frac{2\sqrt{N}}{mM} + \frac{N}{M^2}\right] \ll 1.$$

The above equation suggests that the granularity of the subimages be fairly large if scalability of the algorithm is to hold.

### *Intermediate Level Processing*

The image is mapped onto the nodes of the hypercube array in exactly the same manner as in low level processing. In fact, in many cases intermediate level processing directly follows low level processing and the image is already mapped in that manner. The two factors that affect the perfect scalability of the algorithm are node efficiency and communication overhead.

The subimages, which are of equal areas require, in general, unequal amounts of processing times in the case of intermediate level algorithms. This can be easily seen in the example of the edge following algorithm where each subimage contains a different number of edge pixels. This uneven workload results in some processing nodes finishing the algorithm before other processing nodes and, consequently, waiting idle for them. This in turn causes the node efficiency to drop and affects the perfect scalability of the algorithm. In [21], a Binomial stochastic model is assumed for the distribution of the edge pixels in the image. The model assumes that the probability of a pixel being on an edge is a constant $p$ for all pixels in the image. The drop in efficiency is then quantified. The result in shown in Figure 28 for various values of $p$ as a function of the number of pixels per subimage. As the number of processing nodes grows, the number of pixels per subimage decreases and the node efficiency drops. It might be necessary to re-map the image based on an equal load basis rather than on an equal size basis to improve the node efficiency. Further research is needed to evaluate possible remapping strategies.

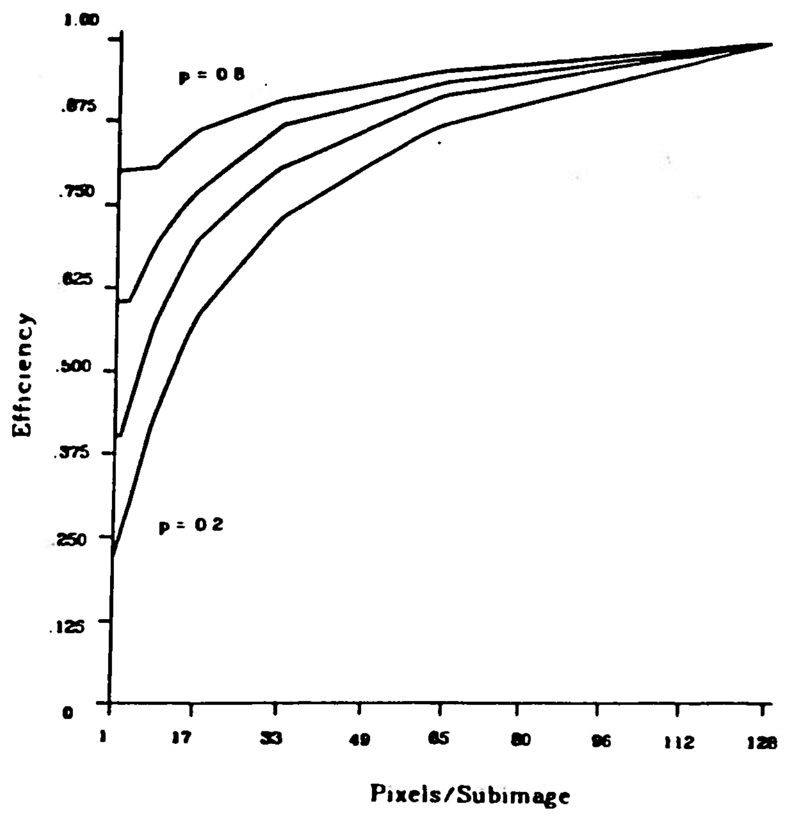


FIGURE 28 Node efficiency for intermediate level algorithms.

The communication overhead for intermediate level algorithms is, in general, difficult to estimate without knowledge of the distribution of the edge pixels in the image. However, intermediate level algorithms generally manipulate less pixels than low level algorithms and, hence, less pixels have to be communicated among the processing nodes. Therefore, equation 4 can form an upper bound on the communication overhead. However, further research still has to be conducted to accurately estimate the communication overhead for intermediate level algorithms.

### *High Level Processing*

The implementation of some of the backtracking search algorithms for high level processing can be, in general divided into three phases: startup phase, computation phase and wind-down phase[22]. In the startup phase, the problem is assigned to a single node in the hypercube array. The problem is then expanded in that node and diffused to other nodes in the hypercube array. In the computation phase, the processors are busy performing the necessary computations. Finally, in the wind-down phase, the results are collected from the hypercube nodes and are combined to obtain the final result.

The node efficiency depends to a large extent on the ratio between the amount of time spent in the computation phase and the time spent in the other two phases. Research has yet to be conducted to quantify this ratio and, hence, the node efficiency.

The communication time overhead depends on the time spent in the startup and wind-down phases as well as whether the node processors communicate during the computation phase to maintain a balanced load of subproblems in the processing nodes. Again, further research is needed to determine the communication overhead.

## CONCLUSIONS

In this chapter a study of special purpose architectures for Robot Vision was presented. The study was composed of two parts. In the first part, algorithms for Robot Vision were classified and the basic characteristics of each class were discussed. Benchmark algorithms for each class were proposed. Although the benchmark algorithms are only a sample of the possible algorithms used in Robot Vision, they do reflect the basic characteristics of each class of these algorithms.

In the second part of the study, special purpose architectures for Robot Vision were discussed. The basic features and characteristics of each group of architectures were presented. Furthermore, the performance of each group of architectures on the benchmark algorithm was presented.

While commercially available machines provide a reasonable solution to some low level processing, they do not provide enough computational power nor do they provide the desired functionality for intermediate and high level processing. It appears for future Robot Vision application that Massively Parallel machines will be needed to provide that power and functionality. However, considerable research has still to be conducted to determine how best to use these machines for higher level processing.

## REFERENCES

1. M. Brady, *Computational Approaches to Image Understanding, Computing Surveys*, vol. 14, no. 1, pp. 3–71, March 1982.
2. E.R. Davis, *Image Processing — its Milieu, its nature, and Constraints on the design of Special Architectures for its Implementation*, in *Computing Structures for Image Processing*, M.J. Duff, ed., New York: Academic Press, 1983.
3. P.E. Danielsson and S. Levialdi, *Computer Architectures for Pictorial Information Systems, Computer*, vol. 14, no. 11, pp. 53–67, Nov. 1981.
4. A. Rosenfeld and A.C. Kak, *Digital Picture Processing*, New York: Academic Press, 1976.
5. R.C.Gonzalez and P. Wintz, *Digital Image Processing*, Reading: Addison-Wesley, 1977.
6. J.F. Canny, *Finding Edges and Lines in Images*, Master Thesis, Department of Electrical Engineering and Computer Science, MIT, June 1983.
7. D.H. Ballard and C.M. Brown, *Computer Vision*, New Jersey: Prentice-Hall, 1982.
8. J.L. Turney, T.N. Mudge and R.A. Volz, *Recognizing Partially Occluded Parts, IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 4, pp. 410–421, July 1985.
9. B.A. Nadel, *The Consistent Labeling Problem, Part 1: Background and Problem Formulation*, Report DCS-TR-164, Computer Science Dept., Rutgers University, New Brunswick, N.J., 1985. Also appears as Report CRL-TR-13-85, Dept. Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 1985.
10. B.A. Nadel, *The Consistent Labeling Problem, Part 2: Subproblems, Enumerations and Constraint Satisfiability*, Report DCS-TR-165, Computer Science Dept., Rutgers University, New Brunswick, N.J., 1985. Also appears as Report CRL-TR-14-85, Dept. Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 1985.
11. B.A. Nadel, *The Consistent Labeling Problem, Part 3: The Generalized Backtracking Algorithm*, Report DCS-TR-166, Computer Science Dept., Rutgers

University, New Brunswick, N.J., 1985. Also appears as Report CRL-TR-12-85, Dept. Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.
12. R.A. Rutenbar, T.N. Mudge and D.E. Atkins, *A Class of Cellular Architectures to Support Physical Design Automation, IEEE Trans. on CAD of IC's and Systems*, vol. CAD-3, no. 4, pp. 264–278, Oct 1984.
13. R.M. Lougheed and D.L. McCubbrey, *The Cytocomputer: A Practical Pipelined Image Processor, Proc. 7th Annual Symp. on Computer Architecture*, pp. 271–277, May 1980.
14. S. Krishna and R. Frisch, *Array Processor Tamed by Structural Innovations, Electronic Products Magazine*, Aug. 1984.
15. G.H. Barnes, et al., *The Illiac IV Computer, IEEE Trans. Computers*, vol. C-17, no. 8, pp. 746–757, Aug. 1968.
16. K.E. Batcher, *Architecture of a Massively Parallel Processor, Proc. 7th Annual Symp. on Computer Architecture*, pp. 168–174, May 1980.
17. D.H. Schaefer, *A Pyramid of MPP Processing Elements — Experiences and Plans, Proc. 18th Int'l Conf on System Sciences*, 1985.
18. H.J. Siegel, et al., *PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition, IEEE Trans. Computers*, vol. C-30, no. 12, pp. 934–947, Dec. 1981.
19. G.F. Pfister, et al., *The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, Proc 1985 Int'l Conf on Parallel Processing*, pp. 764–771, Aug. 1985.
20. T.N. Mudge, *Vision Algorithms for Hypercube Machines, Proc of the IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp. 225–230, Nov. 1985.
21. T.N. Mudge and T.S. Abdel-Rahman, *Efficiency of Feature Dependent Algorithms for the parallel Processing of Images, Proc. 1983 Int'l Conf on Parallel Processing*, pp. 369–373, Aug. 1983.
22. B.W. Wah, G.J. Li and C.F. Yu, *Multiprocessing of Combinatorial Search Problems, Computer*, vol. 18, no. 6, pp. 93–108, June 1985.
23. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Englewood Cliffs: Prentice-Hall, 1975.
24. T.Y. Feng, *A Survey of Interconnection Networks, Computer*, vol. 14, no. 12, Dec. 1981.
25. R.O. Duda and P.E. Hart, *Use of the Hough Transform to Detect Lines and Curves in Pictures, Comm. ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972.
26. S.L. Tanimoto, *A Pyramidal Approach to Parallel Processing, Proc. 10th Annual Int'l Symp. on Computer Architecture*, Stockholm, Sweden, pp. 372–378, June 1983.