Engineering Physics and Mathematics Division

Computing the Hough Transform on an MIMD Hypercube*

Kevin W. Bowyer
University of South Florida
Tampa, FL

Judson P. Jones
Oak Ridge National Laboratory
Oak Ridge, TN

Christopher H. Lake
University of South Florida
Tampa, FL

Paper to be presented at the 6th Scandanavian Image Analysis Conference in Oulu, Finland, June 19–22, 1989

* Research supported by the University of South Florida by Air Force Office of Scientific Research grant AFOSR-89-0036 and National Science Foundation grant IRI-8817776 and at Oak Ridge National Laboratory by the Office of Nuclear Energy, the Office of Technology Support Programs and the Office of Basic Energy Science, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

Computing the Hough Transform on an MIMD Hypercube [1]

Kevin W. Bowyer
Department of Computer Science and Engineering
University of South Florida
Tampa, Florida 33620
kwb@usf.edu

Judson P. Jones
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6364
jov@stc10.ctd.ornl.gov

Christopher H. Lake
Department of Computer Science and Engineering
University of South Florida

Abstract. The computational expense of Hough transforms has prompted researchers to investigate the feasibility of parallel implementations. Most such work to date has dealt with special-purpose architectures or with implementations for SIMD machines such as the MPP or GAPP. This report considers the problem of efficiently implementing Hough transforms on an MIMD hypercube architecture. Beginning with a general analysis of how the data structures might be partitioned to allow a parallel computation, we formulate algorithm and data structure partitioning strategies appropriate to the architecture and then discuss modifications to optimize the performance for a particular hypercube system (the NCUBE). We also present the results of implementing and benchmarking Hough transforms on a 64-processor NCUBE.

1 Introduction

Since its introduction over 25 years ago, the Hough transform has become an important and widely-used tool in computer vision. In view of the large volume of literature dealing with Hough transforms, we do not attempt any comprehensive review here. Instead, we direct the interested reader to two recent survey articles [10, 22].

Due to the computational expense of Hough transforms, researchers have naturally been interested in the possibility of parallel implementation. Illingworth and Kittler [10] cite eleven studies concerned with parallel implementations. Most of these studies focus on single-instruction multiple-data (SIMD) mesh architectures [3, 5, 13, 18, 21], others discuss special purpose and/or VLSI architectures [1, 4, 8, 16], one deals with an SIMD hypercube [14] and another with a switch-based multiple-instruction multiple-data (MIMD) architecture [15]. Wahl [22] cites a number of further studies on special purpose and/or VLSI architectures [2, 7, 12, 19, 20], and with tree-structured SIMD machines [9]. In addition, Rosenfeld [17] presents some initial results of the first DARPA Image Understanding Architectures (IUA) benchmark, of which the Hough transform is one task, and initial results of implementing the Hough transform on an MIMD hypercube are discussed in [11]. The purpose of this report is to present a more detailed analysis of the problems in efficiently implementing the Hough transform on an MIMD hypercube architecture.

[1] Research supported by the University of South Florida by Air Force Office of Scientific Research grant AFOSR-89-0036 and National Science Foundation grant IRI-8817776 and at Oak Ridge National Laboratory by the Office of Nuclear Energy, the Office of Technology Support Programs and the Office of Basic Energy Science, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

2 The Hough Transform Benchmark

The general sequence of steps involved in applying the Hough transform is: 1) detect edges in the original image to obtain a binary image where each pixel is marked as **edge** or **not edge**, 2) compute the transform itself: for each **edge** pixel, increment a set of accumulator bins in Hough space which correspond to possible parameterizations of a shape on which the edge pixel would be found, 3) select local maxima in Hough space, which should correspond to those instances of a parameterized shape (canonically, straight lines) which actually appear in the image. Since the number of possible instances of a shape which could give rise to the observed edge pixel is typically large, the computation of the transform itself (step 2), is generally much more expensive than the other two steps. Consequently, we will concentrate on this step.

The two major data structures involved in computing the Hough transform are: 1) the binary image of edge pixels and 2) the Hough space. Computing the transform can be viewed as a (possibly interleaved) two-stage process. First, the binary image must be scanned for edge pixels. Second, for each edge pixel, the appropriate locations in Hough space must be incremented.

As a first example, let us consider the Hough transform for lines as defined in the IUA benchmark. In this benchmark, a 512×512 binary image is assumed as input, with the origin assumed to lie in the lower left corner. The "votes" contributed by each edge pixel are accumulated into a two-dimensional "ρ, θ" Hough space. For each edge pixel, a value of ρ is computed for each value of θ in the range of 0 to 179, where ρ is the distance along a line at angle θ from the origin to a perpendicular line which runs through the edge pixel. Each such ρ, θ location in Hough space is incremented. (See Figure 1.)

As a second example, we consider the Hough transform for a particular parametrized shape. The detection of rectangular outlines is simple and will serve to illustrate the relevant issues. If rectangular outlines appearing in the image are aligned with the X and Y axes of the image (as might be the case for box-shaped obstacles standing on the floor), then they may be parametrized by their location in the image, their height and their height/width ratio. If the binary image is of dimension 512×512, the number of distinct heights is H, and the number of distinct widths for each height is W, then the dimension of the Hough space is $512 \times 512 \times H \times W$. For each edge pixel, a vote is accumulated for each possible location and size of a rectangular outline which would contain this pixel. If the average height value is h and the average width value is w, then the number of votes accumulated for a given edge pixel is $H \times W \times h \times w$. (See Figure 2.) In the computation we have implemented, the height/width ratio is fixed at one, giving square outlines, and the height ranges over the ten values [41,43,...59], giving us a $512 \times 512 \times 10$ Hough space.

These two Hough transforms were implemented in C on a SUN workstation and on a 64-processor NCUBE hypercube, and timings were obtained for a set of test images. Four "synthetic" test images were used, with 100, 1000, 10000, and 30000 randomly distributed edge points (similar to the 30,000-point image described in [15]). One real image (of San Francisco) was used, with a small number of randomly chosen edge points removed so that exactly 30000 edge points remained. The results of the computations on the SUN were compared with those obtained on the NCUBE in order to verify that the parallel computation was correct.

3 Algorithm Partitioning

The most straightforward method of assigning computations to processing elements in an MIMD architecture is to use a "single program - multiple data stream" (SPMD) model. In an SPMD model, a complete copy of the same program resides in the local memory of each processing element (PE), but operates with possibly different threads of execution on different data structures or on different portions of a distributed data structure. Interprocessor communication synchronizes these programs, which as a result tend to cleave naturally into a sequence of alternating epochs of (possibly overlapped) computation and communication. In the SPMD model, the key to efficiency is in distributing the data structures so as to simultaneously 1) balance the computational workload across PEs, avoiding redundant calculations and 2) minimize delays associated with communications between PEs. An ideal SPMD application is one in which all PEs execute an identical number of distinct calculations with no communication. Since this is rarely possible, the practical objective is to balance the computation by distributing data among all PEs such that each PE executes approximately the same number of instructions, and minimize communication delays by minimizing the number of communication epochs (synchronization points), the number of messages passed in each communication epoch, and the length of each message. For any given machine, there are specific costs associated with the transmission of any message (setup time)

and the message length (communication rate). Also, for any given application, there are specific costs associated with unbalanced computations (delays in subsequent synchronization).

4 Logical Partitioning of Access to the Data Structures

We first consider how access to the data structures may be logically partitioned among PEs. At the highest conceptual level, access to each data structure may be either **complete**, meaning that each PE accesses (at least potentially) the entire data structure, or **partitioned**, meaning that each PE accesses a restricted portion of the data structure. Each PE in an MIMD hypercube system has its own purely local memory, in the sense that the addresses generated by different PEs refer to distinct memory spaces. Therefore, **complete**-access implies that the entire data structure must be replicated in each PE's local memory, whereas **partitioned**-access implies a distribution of the data structure such that one partition resides in each PE's local memory.

This distinction between **complete-** and **partitioned-access** data structures yields four basic alternatives:

1. **complete** binary image / **complete** Hough space.
   Each PE scans the entire binary image and updates all appropriate elements of Hough space for each edge pixel. In other words, each PE individually executes a complete sequential computation of the transform. The amount of work done by each PE is, obviously, evenly balanced across PEs, and there is no communication required between PEs. But because the calculations are all redundant, this alternative generally does not make sense – there is no speedup over a single PE.

2. **complete** binary image / **partitioned** Hough space.
   Each PE scans the entire binary image, but accumulates votes only within its assigned partition of Hough space. Since each PE looks at the entire image, the image scan part of the computation is well-balanced, albeit redundant. If the Hough space is partitioned appropriately, then the computation for accumulating votes should also be reasonably well-balanced. Again, no communication is required between PEs. However, a large amount of memory is required at each PE. For some hypercube systems (such as the original NCUBE, with 128K bytes per PE, or the current NCUBE, with 512K bytes per PE) this may present an obstacle which effectively eliminates this alternative. For other hypercube systems (such as the current Intel iPSC, with up to 16M bytes per PE) this should not present a problem.

3. **partitioned** binary image / **complete** Hough space.
   Each PE scans its assigned partition of the image for edge pixels, and accumulates votes to all of Hough space for each edge pixel. If the number of edge pixels per PE is approximately the same (see below), then the computation is well-balanced and not redundant. But since each PE operates with its own local copy of Hough space, this does not, by itself, result in a completed Hough space at any PE. To complete the computation, either 1) image partitions must be exchanged and the local Hough spaces updated for each partition, or 2) the local Hough spaces must be accumulated globally. The first alternative is of no merit, since updating the local Hough spaces requires redundant calculation – eventually all PEs compute the entire Hough space, and the whole calculation is as slow as in the sequential case. The second alternative is more reasonable – local Hough spaces can be accumulated onto a single node using a logarithmic minimum spanning tree, with a cost proportional to $O(logP * HS)$, where P is the number of PEs and HS is the size of Hough space. However, since each PE may process radically different numbers of pixels, the initial computation of the local Hough spaces will likely be very unbalanced.

4. **partitioned** binary image / **partitioned** Hough space.
   Each PE scans its assigned image partition and accumulates votes to its assigned partition of the Hough space. By itself, this results in only a partial computation. To complete the computation, this step must be repeated while PEs cycle through the partitions of either the image or the Hough space. (Each pixel contributes to possibly many partitions of the Hough space.) Exchanging partitions of the image is usually the better choice, since the communication cost is determined by the (usually smaller) number of edge pixels rather than by the size of the Hough space.

In both the **partitioned** binary image/**complete** Hough space and the **partitioned** binary image/**partitioned** Hough space alternatives, an approximately balanced workload can be guaranteed with a logarithmic cost pre-processing step in which edge pixels are exchanged so that the number of

edge pixels on each PE is equalized. At successive stages of the exchange, pairs of directly connected PEs along successive dimensions of the hypercube exchange the number of edge pixels currently residing on each PE. Then one PE sends its "excess" edge pixels to the PE with the smaller number. At the end of this process, all PEs have approximately the same number of edge pixels. (See Figure 3). Specifically, the difference between the number of edge pixels contained on two PEs is no more than the Hamming distance between the two PEs, so that the residual imbalance (defined for this application as the largest such difference for any pair of PEs) is no more than the dimension of the hypercube. For a 30,000 edge point Hough transform on a 64 PE hypercube, this is no more than 1.28 percent (6 points).

From this analysis, it appears that there are really just two feasible alternatives. One is to structure the parallel computation with a replicated complete binary image at each PE, and a Hough space partitioned across PEs. Due to memory requirements, this option might be feasible on a system like the Intel iPSC but not feasible on a system such as the NCUBE. The other feasible alternative is to structure a parallel computation with both data structures partitioned across PEs, building lists of edge pixels and exchanging them across the hypercube in order to complete the computation. This alternative should be feasible on systems with even a very small amount of memory per PE, and since the computation and communication can be overlapped to some extent, it should be possible to "hide" some of the communication costs.

5 Mapping Partitions Onto Physical Memory

The question that now remains is how exactly to partition the data structures for mapping onto the hypercube. Assume that there are P PEs and an $N \times N$ image. If $P > N^2$, then some PEs necessarily go unused, since there will be more PEs than pixels in the image. If $N < P \leq N^2$, partitions of less than one row of elements are required in order to make use of all the PEs, and square partitions are preferred. In instances where $P \leq N$, it is possible to partition the image into either blocks of complete rows (or columns) or into general square subregions. For a given P and N, the square partition has fewer border elements than does the block of complete rows partition. However, it is sometimes the case that the block of complete rows partition will result in a more efficient algorithm. The reasoning for this is as follows.

Consider applying a local neighborhood operator, such as the 3×3 Sobel edge detector, to the image. When the image is partitioned into blocks of complete rows, then each partition has just two neighboring partitions and they are easily mapped onto directly connected PEs of the hypercube in an embedded graycode ring [11]. However, when square partitions are used, it is not possible for neighboring partitions to be mapped onto directly connected PEs. To see why this is true, consider the computation for a corner element in a general square partition. This computation requires values to be communicated from three adjacent partitions. Directly connected PEs have, by definition of the hypercube, PE numbers which differ in exactly one bit position. For all the neighboring partitions of the corner element to lie on directly connected PEs, it would have to be the case that three overlapping pairs of PE numbers each differ in exactly one bit, which cannot be true (there are no "odd cycles" in a hypercube). (See Figure 4.)

The result is that a block of complete rows partition incurs a substantially different communication cost than a general square partition. Which mapping should actually be used depends on the relative costs of different components of the communication. For particular values of the different components of the communication cost which approximate those found in the current NCUBE, mapping blocks of complete rows results in a lower communication cost than mapping square subregions. This is a pleasing result, since there are other strong practical reasons for using a mapping of blocks of complete rows. One is that the code to implement the message passing is much simpler. Another is that the code to load/unload image data into/from the hypercube is also much simpler. Since the binary edge image is derived from the original grayscale image by applying a local operator, it is natural to use the same partitioning for the binary edge image as for the original grayscale image.

In the case of the transform for lines, partitioning the Hough space along the θ dimension results in the best performance. This is because each edge pixel generates votes to Hough space for a range of 180 distinct θ values. Thus a uniform partitioning along the θ dimension results in a workload which is evenly balanced across PEs and where each PE's computation is independent of all others. If the partition of the θ dimension is on an embedded graycode ring, subsequent processing for finding local maxima using image processing algorithms is eased.

In the case of the transform for rectangles, partitioning Hough space for a balanced computation is slightly more difficult. Partitioning along one of the dimensions representing the location in the image

may result in a highly unbalanced computation, since then some PEs may have no work for some edge pixels. In this case, partitioning along a dimension which represents one of the other parameters should result in a better load balance. However, this partitioning has a built-in imbalance to whatever degree the size of the outline varies with the parameter value. In our case, partitioning Hough space along the height dimension ensures that each PE performs some of the computation generated by each edge pixel. However, the partitions representing greater height values generate a greater computational load than the partitions representing lesser height values, since the outline is larger and so requires more votes. Splitting the slice of the Hough space for a given height value across PEs is not a solution, since this introduces required communication between PEs. Fortunately, the load imbalance inherent in the partitioning of Hough space should, in most cases, be relatively small.

6 Benchmarks on the NCUBE Hypercube System

We implemented the Hough computation using partitioned image and Hough space on the 64-PE NCUBE system in the CESAR lab at Oak Ridge National Laboratory. The NCUBE uses custom VLSI processors with hardware floating-point and has a 512 KB local memory for each PE. The NCUBE/10 can be configured with up to 1,024 PEs, and a near linear speedup over this entire range has been demonstrated for some engineering computation problems [6]. The main language constructs for controlling parallelism in the NCUBE version of C are *nwrite* and *nread*, which are used to send and receive messages, *ntest()*, which is used to test if there is a message in the message buffer, and *whoami()*, which is used to determine the physical number of a PE in the hypercube.

The 'preferred' version of the parallel computation proceeds as follows. Each PE is initialized with the same environment, and the image partitions are mapped onto the PEs of the hypercube using a graycode ring. First, each PE scans its partition of the image and forms a list of the edge pixels. Then the lengths of the lists are equalized through a sequence of exchanges along the dimensions of the hypercube, as explained earlier. Finally, the transform itself is computed through another succession of exchanges along dimensions of the hypercube, interleaved with incremental accumulation of votes to the Hough space. In each of the successive exchanges, each PE 1) initiates an *nwrite* of its current pixel list, 2) updates its partition of Hough space for the 'new' pixels in its pixel list, and 3) performs an *nread* to accept the pixel list from its neighbor on this dimension and adds these 'new' edge pixels to its current pixel list. This ordering of steps in each exchange allows computation to be overlapped with message passing to whatever extent possible. The lists exchanged along the first dimension are roughly N/P in length, where N is the total number of edge pixels in the image and P is the number of PEs, and the lists exchanged along the final dimension are roughly $N/2$ in length. In this way, after exchanges along all dimensions, each PE has updated its partition of Hough space for all edge pixels in the image.

Table 1 summarizes benchmark timings (in milliseconds) for this 'preferred' version of the computation, along with seven other versions of the computation, using the transform for lines and the synthetic 1,000 edge pixel image. Times for the preferred version of the computation are listed in line 1. Line 2 represents a version of the computation which is the same except for not overlapping the computation and message passing in computing the transform. Lines 3 and 4 represent the same computations as lines 1 and 2, respectively, but without the pre-processing step in which the lengths of the edge pixel lists at each PE are equalized. Lines 5 through 8 represent the same computation as lines 1 through 4, respectively, but with each PE forwarding its list of edge pixels along the graycode ring in computing the transform (using $N-1$ message steps) instead of forwarding cumulative lists along the cube (using $log_2 N$ message steps). Note that, other factors being equal, the computation using overlap is always faster and the amount by which it is faster increases with the dimension of the hypercube. Also, other factors being equal, cube forwarding of pixel lists is always faster than ring forwarding and the amount by which it is faster increases with cube dimension. These effects are both as would be expected. However, the use of load balancing results in a *slower* computation. This is because a balanced workload was already assured by the way in which the synthetic image was generated, and so the load balancing represents an unnecessary overhead. However, we still expect that for real images, where the initial edge pixel distribution may be very uneven, load balancing will prove to be worthwhile.

Two major problems arose in benchmarking the 'preferred' version of the computation for the originally chosen set of benchmark images. Both problems relate to the memory limitations of the NCube system. The first problem is that, due to the size of the message buffers kept by the operating system, it was not possible to use cube forwarding for images with larger numbers of edge pixels. For this reason, we used the ring forwarding version of the computation for a benchmark computation to compare across all the images. The second problem is that storing one 512×512 slice of Hough space, representing one

possible value for rectangle height, takes 256K of memory, which is half of the available memory in a PE. Thus it is clearly not possible to store more than one slice of Hough space at a PE. If we want each PE to operate on an integral number of slices, then we can only use of a number of PEs equal to the number of height values. Also, the $512 \times 512 \times 10$ Hough space could not be stored in a hypercube of less than dimension three. Thus, for our benchmark computation, we simply computed one slice of the Hough space (for a height value of 41), and partitioned the Hough space by blocks of rows. In an application, different slices of the Hough space could be computed iteratively, with local maxima detected inbetween computing successive slices. The actual benchmark computation exhibits a speedup function which falls noticeably below that for the transform for lines.

The results of the final benchmark computations for lines and rectangles are summarized in Figures 5 and 6. The reported execution times do not include the time needed to read the initial image data structure into the hypercube. Several general conclusions can be drawn from the benchmarks. With the parallel Hough computation for lines, a nearly linear speedup is possible, provided that the amount of work to divide up between the PEs is large in comparison to the overhead. Images with a smaller number of edge pixels exhibit a relatively poor speedup function. Also, the initial distribution of edge pixels in the image can have a noticeable effect on the observed speedup (compare the speedup for the real and synthetic 30,000-point images).

7 Summary

It is possible to formulate essentially "fully parallel" Hough transform computations on an MIMD hypercube because 1) the binary image can be scanned by all PEs in parallel, and 2) the Hough space can generally be partitioned along some dimension which distributes the workload fairly evenly between PEs and allows updates of Hough space without potential access conflicts. Assuming that the percentage of edge pixels in an image is independent of image size, the complexity of the sequential Hough transform is $O(N^2 \times V)$, where V is the number of votes generated by each edge pixel. Our parallel formulation, then, essentially divides V by the number of PEs. In the case of the transform for lines, the parallel algorithm could be expected to make reasonable use of up to 180 PEs (one for each value along the θ dimension). However, several factors contribute to degrading the observed actual speedup, chief among them the total workload of the computation. As the absolute size of the workload executed by each PE declines, the communication overhead becomes a larger relative portion of the total execution time.

The main difference between the ρ, θ Hough transform for lines and the Hough transform for rectangles described here is that, in the former, votes from each pixel are evenly distributed across the θ dimension whereas in the latter the vote distribution is more localized. This localization complicates the problem of partitioning the data structures so as to achieve a balanced workload. In the transform for lines, partitioning Hough space into uniform blocks of θ values and cycling all edge pixels by each PE simultaneously guarantees that 1) all PEs perform the same amount of work, and 2) each PE's access to Hough space is independent of all others. In the transform for rectangles, there is no way of partitioning Hough space so as to simultaneously achieve both goals. If Hough space is partitioned by complete slices of the height parameter, then each PE's access to Hough space is independent of all others but the amount of work done by each PE varies. If Hough space partitions are allowed to begin/end within a height value, then a more balanced workload can be achieved, but different PEs will sometimes need to access the same elements of Hough space. For hypercube machines, it will generally be preferable to have a modest load imbalance rather than to introduce extra communication between nodes. For the NCUBE specifically, memory limitations caused us to partition the Hough space for a single height value across all PEs, thereby accepting a possibly large load imbalance.

The implementations described here heavily utilize embedded graycode rings for both image and Hough space domain decompositions. For the convolution-like image processing routines likely to be used in pre- and post-processing, these decompositions are optimal for only a restricted set of image and Hough space sizes, and this optimality depends on certain machine parameters such as the overhead accrued in initiating communication and the rate at which data are communicated between adjacent processors. For large images and/or large hypercubes, rectangular domain decompositions theoretically yield better performance. However, the graycode ring decomposition for the Hough space for lines is arguably optimal, since for each value of θ a pixel may contribute to any value for ρ, and subdividing the ρ dimension will necessarily result in redundant excess calculation (a PE must decide if a given pixel will contribute to its section of ρ). For post-processing, the conversion of a ring-mapped Hough space into a suitably grid-mapped Hough space requires only $\Theta(log_2\sqrt{P})$ messages, and both the complexity of the post-processing and the machine parameters will determine whether this cost is worth paying.

References

[1] Baringer, W.B., Richards, B.C., et al. 1987. A VLSI Implementation of PPPE for real time image processing in Radon space, CAPAMI Workshop, 88-93.

[2] Chuang, H.Y.H., and Li, C.C. 1987. A Systolic Array Processor for Straight Line Detection by Modified Hough transform, CAPAMI Workshop, 300-304.

[3] Cypher, R.E., Sanz, J.L.C., and Snyder, L. 1987. The Hough transform has O(N) complexity on SIMD NxN mesh array architectures, CAPAMI Workshop, 115-121.

[4] Fisher, A.L. and Highman, P.T. 1987. Computing the Hough transform on a scan line array processor, CAPAMI Workshop, 83-87.

[5] Guerra, C. and Hambrusch, S. 1987. Parallel algorithms for line detection on a mesh, CAPAMI Workshop, 99-106.

[6] Gustafson, J.L., Montry, G.R. and Benner, R.E. 1988. Development of Parallel Methods for a 1024-processor Hypercube, SIAM J. Sci. Stat. Comp. 9, 4 (July 1988).

[7] Hanahara, K., Maruyama, T. and Uchiyama, T. 1986. High-Speed Hough Transform Processor and its Applications to Automatic Inspection and Measurement, IEEE Robotics and Automation Conference, 1954-1959.

[8] Hanahara, K. ,Maruyama, T. and Uchiyama, T. 1988. A real-time processor for the Hough transform, IEEE PAMI 10, 1, 121-125.

[9] Ibrahim, H.A.H., Kender, J.R., and Shaw, D.E. 1986. On the Application of Massively Parallel SIMD Tree Machines to Intermediate-Level Vision Tasks, CVGIP 36, 53-76.

[10] Illingworth, J. and Kittler, J. 1988. A Survey of the Hough transform, CVGIP 44, 1 (October 1988), 87-116.

[11] Jones, J.P. 1988. A Concurrent On-Board Vision System for a Mobile Robot, Third Conference on Hypercube Concurrent Computers and Applications, 1022-1032, Pasadena, California.

[12] Kung, H.T., and Webb, J.A. 1985. Global Operations on a Systolic Array Machine, IEEE Int. Conf. on Comp. Design., 165-171.

[13] Li, H. 1985. Fast Hough transform for multidimensional signal processing, IBM research report RC 11562, Yorktown Heights.

[14] Little, J.J., Blelloch, G. and Cass, T. 1989. Algorithmic techniques for computer vision on a fine-grained parallel machine, IEEE PAMI 11, 3, 244-257; See also ICCV '87, 335-340.

[15] Olson, T.J., Bukys, L. and Brown, C.M. 1987. Low level image analysis on an MIMD architecture, ICCV '87, 468-475.

[16] Rhodes, F.M., Dituri, J.J., et al. 1988 A monolithic Hough transform processor based on restructurable VLSI, IEEE PAMI 10, 1, 121-125.

[17] Rosenfeld, A. 1987. A Report on the DARPA Image Understanding Architectures Workshop, Proc. DARPA Image Understanding Workshop, 298-302.

[18] Rosenfeld, A., Ornelas, J., and Hung, Y. 1988. Hough transform Algorithms for Mesh-Connected SIMD Parallel Processors, CVGIP 41, 3 (March 1988), 293-305.

[19] Sanz, J.L.C. and Dinstein, I. 1987. Projection-Based Geometrical Feature Extraction for Computer Vision: Algorithms in Pipeline Architectures, IEEE PAMI 9, 1, 160-168.

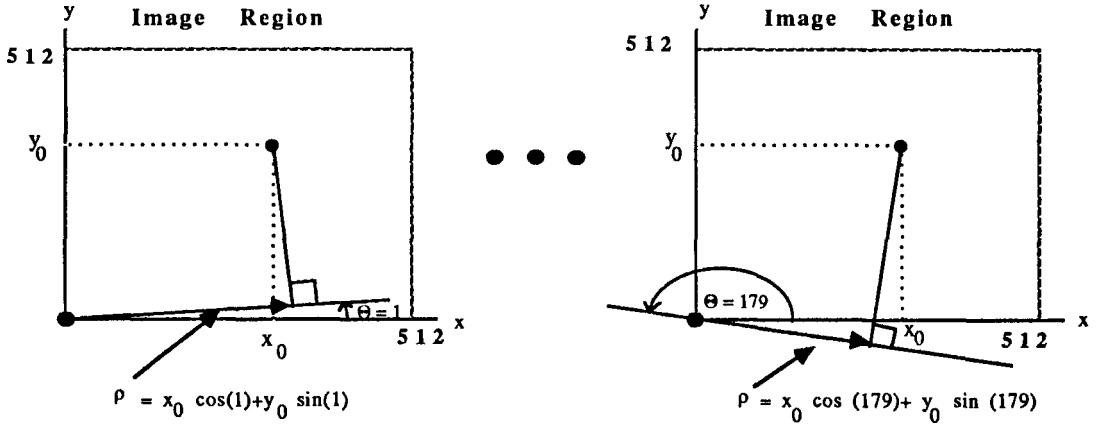
[20] Sher, D. and Tevanian, A. 1984. Vote Tallying Chip: A Custom Integrated Circuit, Univ. of Rochester TR-44.

[21] Silberberg, T.M. 1985. The Hough transform on the GAPP, IEEE Conf. on Computer Architecture for Pattern Analysis and Image Database Management, 387-393.

[22] Wahl, F.M. 1988. A Survey of Hough transform Techniques, Technical Report 5-88-1, Technische Universitat Braunschweig, D-3300 Braunschweig, FRG.

Table 1 - Benchmark Times for Different Versions of Parallel Hough Computation

	options in each version			dimension of hypercube					
	forward	balance	overlap	1	4	3	4	5	6
1.	cube	on	on	4594	2306	1179	616	318	170
2.	cube	on	off	4599	2313	1187	627	331	186
3.	cube	off	on	4592	2303	1175	612	315	165
4.	cube	off	off	4688	2453	1259	648	343	190
5.	ring	on	on	4621	2321	1191	632	346	226
6.	ring	on	off	4626	2326	1200	646	370	272
7.	ring	off	on	4620	2318	1187	628	341	219
8.	ring	off	off	4716	2566	1375	739	433	315



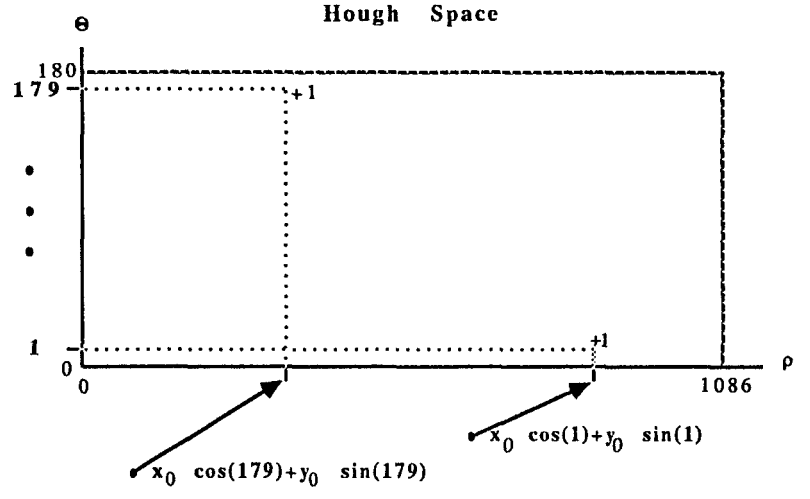


Figure 1 - Computation of the Hough Transform for Lines with ρ-Θ Hough Space. For each edge pixel encountered in scanning the image, 180 votes are accumulated in the ρ-Θ Hough Space.

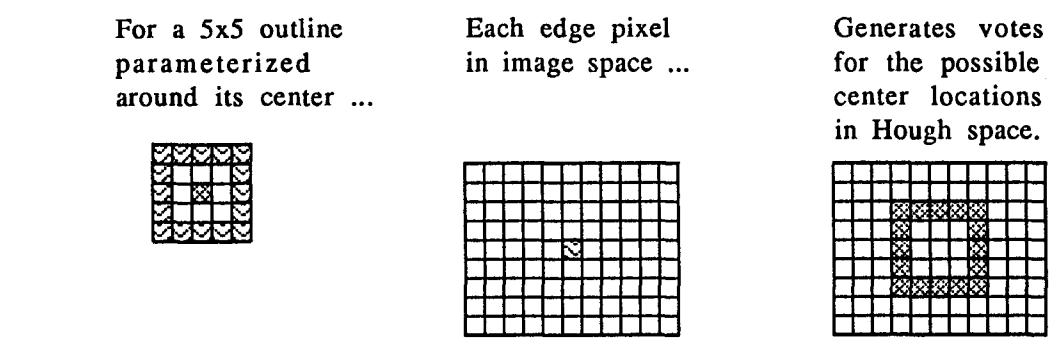
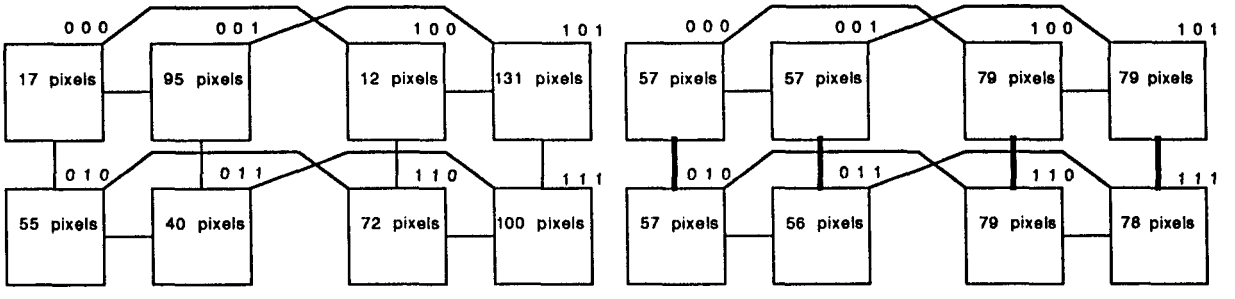


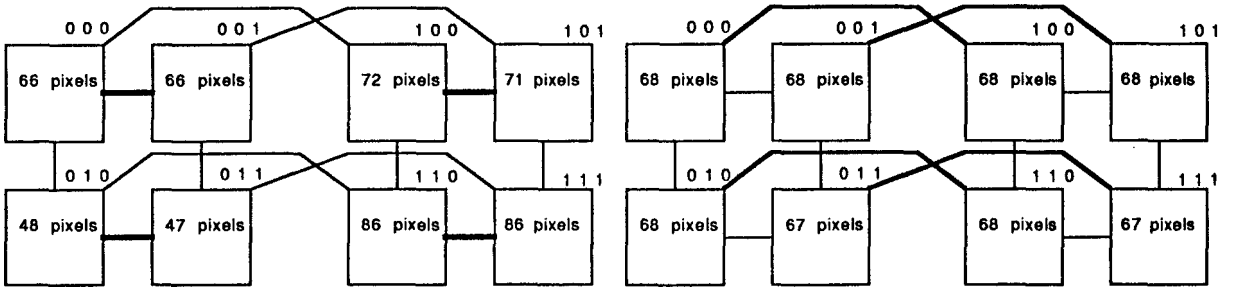
Figure 2 - Example of the Hough Transform for Parameterized Outlines.

The Hough space votes generated for a 5 x 5 square outline are depicted in the figure. The benchmark computation detects outlines in the range of 40 to 49 pixels, resulting in a Hough space of size 512 x 512 x 10.



(a) initial distribution of pixels among nodes

(c) lengths of pixel lists after exchange on dimension 2 (bold lines)



(b) lengths of pixel lists after exchange on dimension 1 (bold lines)

(d) lengths of pixel lists after exchange on dimension 3 (bold lines)

Figure 3 - Example of Balancing Lengths of Edge Pixel Lists Across Nodes.of the Hypercube.

At each stage, the neighbors on one dimension of the hypercube exchange their current edge pixel lists, so that each has a copy of the combined list, and the 0 (1) neighbor then keeps the bottom (top) half of the combined list. After the final exchange, the range of the lengths of the lists will be no greater than the dimensionality of the hypercube

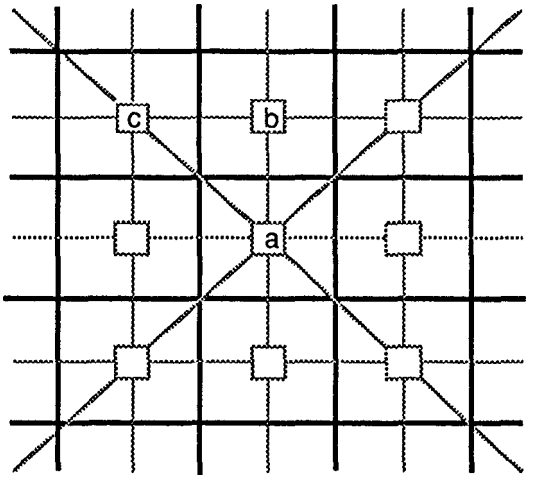


FIGURE 4 - 8-Connected Image Does Not Map Onto Directly Connected Nodes.

In order to embed subregions of the image (bold lines) so that the 8-connected neighbor regions are embedded on directly connected nodes of the hypercube (dotted lines), each of the pairs of node numbers a-b, b-c, and c-a must differ in exactly one bit position, but these three pairs form an odd-length cycle, and so such an embedding is not possible on a hypercube.

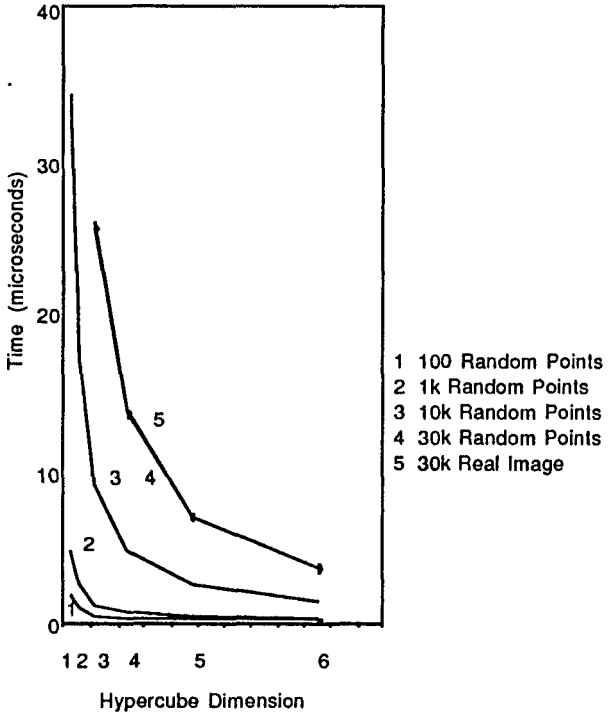


Figure 5 - Hough Transform for Lines

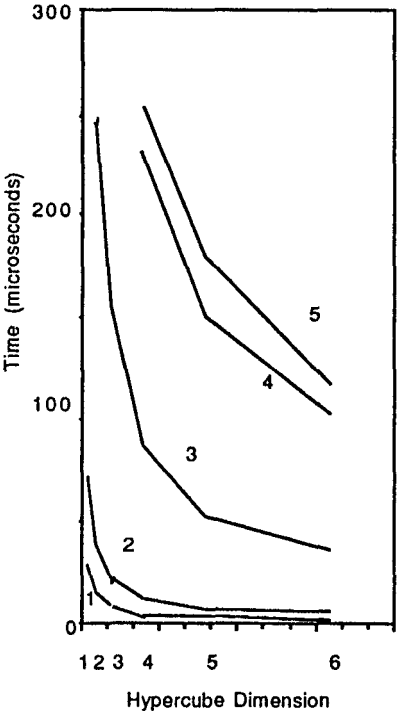


Figure 6 - Hough Transform for Rectangles