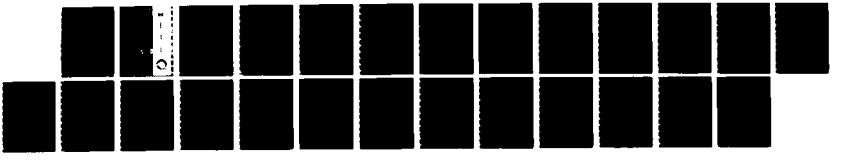
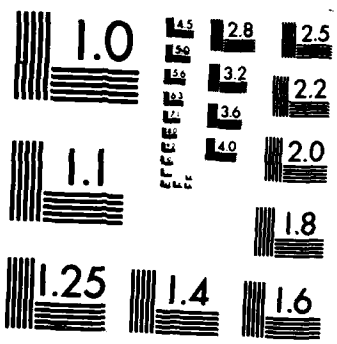
AD-A175 103 THE HOUGH TRANSFORM ON THE BUTTERFLY AND THE NCUBE(U) 1/1
MARYLAND UNIV COLLEGE PARK CENTER FOR AUTOMATION
RESEARCH S CHANDRAN ET AL SEP 86 CAR-TR-226 ETL-0438
UNCLASSIFIED DACA76-84-C-0004 F/G 9/2 NL





CROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ETL-0438

AD-A175 103

The Hough transform on the Butterfly and the NCUBE

Sharat Chandran
Larry S. Davis

Center for Automation Research
University of Maryland
College Park, Maryland 20742

September 1986

Prepared for
U.S. ARMY CORP OF ENGINEERS
ENGINEER TOPOGRAPHIC LABORATORIES
FORT BELVOIR, VIRGINIA 22060-5546

and
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
1400 WILSON BOULEVARD
ARLINGTON, VIRGINIA 22209-2308

E T L

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS AD-A175 103
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CAR-TR-226 CS-TR-1713	5. MONITORING ORGANIZATION REPORT NUMBER(S) ETL-0438
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U.S. Army Engineer Topographic Labs
6c. ADDRESS (City, State, and ZIP Code) Center for Automation Research University of Maryland College Park, MD 20742	7b. ADDRESS (City, State, and ZIP Code) Research Institute Fort Belvoir, VA 22060-5546
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA 8b. OFFICE SYMBOL (If applicable) ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DACA76-84-C-0004
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22209-2308	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62301E; PROJECT NO.; TASK NO.; WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

THE HOUGH TRANSFORM ON THE BUTTERFLY AND THE NCUBE

12. PERSONAL AUTHOR(S)
Chandran, Sharat and Davis, Larry S.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1986, September	25

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
09	02		Parallel implementation, Hough Transform, Butterfly parallel processor, NCUBE

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes the parallel implementation of the Hough Transform, a technique to detect colinear edge points. Specifically, two contrasting architectures, the Butterfly Parallel Processor, essentially a shared memory machine, and the NCUBE, a direct connection machine in which processors are interconnected in the form of a hypercube are considered.
Developing parallel Hough transform algorithms involves addressing questions of optimal processor allocation and parallel "peak" selection in image neighborhoods. Fast, practical algorithms (subject to inherent lower bounds) are presented, and relevant complexity issues are discussed.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Rosalene M. Holecheck	22b TELEPHONE (Include Area Code) (202) 355-2767 22c OFFICE SYMBOL ETL-RI-T

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PREFACE

This document was prepared by the Center for Automation Research at the University of Maryland, College Park, Maryland, under contract number DACA76-84-C-0004 for the U.S. Army Engineer Topographic Laboratories, Fort Belvoir, Virginia, and the Defense Advanced Research Projects Agency, Arlington, Virginia. The Contracting Officer's Representative was Ms. Rosalene M. Holecheck.

CAR-TR-226
CS-TR-1713

DACA76-84-C-0004
September 1986

THE HOUGH TRANSFORM ON THE BUTTERFLY AND THE NCUBE

Sharat Chandran
Larry S. Davis
Center for Automation Research
University of Maryland
College Park, MD 20742

ABSTRACT

This report describes the parallel implementation of the Hough Transform, a technique to detect collinear edge points. Specifically, we consider two contrasting architectures, the Butterfly Parallel Processor[1], essentially a shared memory machine, and the NCUBE[2], a direct connection machine in which processors are interconnected in the form of a hypercube.

Developing parallel Hough transform algorithms involves addressing questions of optimal processor allocation and parallel "peak" selection in image neighborhoods. We present fast practical algorithms (subject to inherent lower bounds) and discuss the relevant complexity issues.

The support of the Center for Automation Research by the U.S. Army Engineer Topographic Laboratories under Contract DACA76-84-C-0004, and of the first author by FMC Corporation, is gratefully acknowledged.

[1]Butterfly is a trademark of Bolt Beranek and Newman, Inc.

[2]NCUBE is a trademark of NCUBE Corporation.

1 Introduction

This paper discusses parallel algorithms and their implementation for the class of operations that involve computing two-dimensional histograms of image features. An important class of such operations are the so-called *Hough Transforms* [DH72], which can be used to detect straight lines or edges (or more generally, arbitrarily shaped point patterns) in an image. This problem is representative of computationally intensive intermediate level vision problems and is one that merits using a machine whose architecture is non Von-Neumann.

This paper is concerned with two machines of very different characteristics. The first is the Butterfly Parallel Processor, a tightly-coupled, shared memory machine. Our programs were implemented on a sixteen node machine at the Center for Automation Research. The other machine is a sixty-four node NCUBE machine available at the University of Maryland's Center for Advanced Research in Biotechnology. The processors in this machine have some local memory and are located at the corners of a hypercube.

The main issue in parallel processing is the efficient use of processors. Let P be the number of processors and let N be a measure of problem size. Since physical multiprocessors will be built with a finite number of processors, it may well be that the user pushes the limits of the machine by taking on a very large problem. (It is also not hard to visualize problems broken into modules and cases where the multiprocessor is essentially partitioned.) Here we have a case where $P \leq N$. It is also important to consider the case when $P \geq N$ since, as Arvind [AI86] points out, one wants processors to be *scalable* in such a manner that adding hardware resources results in increased performance; one wants to analyze the asymptotics of this behavior. Accordingly, it is essential that we discuss independently how the algorithm and the architecture behave as the number of processors and the problem size vary.

In all the cases, the goal is that the time to solve the problem sequentially be P times the time to solve the problem in parallel, but it is not clear whether this can be achieved for all input sizes. If we can achieve this, the parallel algorithm is *optimal.* Let $T_1(N)$ represent the time complexity of the fastest algorithm on a problem of size N using a single processor. Likewise, let $T_P(N)$ be the complexity of a parallel algorithm using P processors. Given any parallel algorithm A for a given problem along with a particular model of computation, then there is a processor efficiency function(PEF) $S(N)$ such that for $1 \leq P \leq S(N)$

$$T_P(N) = O(T_1(N)/P).$$

Given two algorithms A_1 and A_2, and the corresponding PEFs $S_1(N)$ and $S_2(N)$, we say that A_1 is more *processor efficient* than A_2 if there exists N_1 such that for all $N > N_1$, $S_1(N) > S_2(N)$[1]. A subgoal in algorithm design thus is to maximize the PEF and one characterizes an algorithm A by saying that it is in the Class $K_{S(N)}$.

Alternatively, one can consider for any P, the *minimal* input size N for which the algorithm is optimal and define an analogous *data efficiency function* (DEF). Thus, given an optimal algorithm B, there exists a function $D(P)$ such that for all $N > D(P)$,

$$T_P(N) = O(T_1(N)/P)$$

The algorithm B is then said to be in the Class $J_{D(P)}$ and the subgoal is to minimize $D(P)$. Given two algorithms B_1 and B_2 and the corresponding DEFs $D_1(P)$ and $D_2(P)$, we say that B_1 is more *data efficient* than B_2 if $D_1(P) = o(D_2(P))$ [2]. Section 4.1.1 illustrates these concepts.

The remainder of this paper is organized as follows. Section 2 defines the specific type of Hough transform that we are interested in implementing and describes previous work

[1] In other words, $S_1(N) = \Omega(S_2(N))$.

[2] Given functions f and g of x, $f = o(g)$ if $\lim_{x\to\infty}(f/g) = 0$.

done in this area. Section 3 describes the characteristics of the parallel machines we are considering. Section 4 presents the algorithms and discusses the relevant complexity issues pertaining to them as well as some useful implementation strategies. Section 5 contains some concluding remarks.

2 Definition of the Problem

A common problem in computer image processing is the detection of straight lines in digitized images. In the simplest case, the image contains a number of discrete feature points perhaps corresponding to the edge pixels obtained by a local operator (see below). The problem is to detect the presence of groups of collinear or almost collinear feature points. It is clear that the problem can be solved to any desired degree of accuracy by testing the lines formed by all pairs of points. However, the computation required for n points is approximately proportional to n^2, and may be prohibitive for large n.

Rosenfeld [Ros69] describes a method originally proposed by Hough [Hou62] and subsequently improved upon by Duda and Hart [DH72] to solve the problem. Essentially, a Hough transform is designed to detect collinear sets of edge pixels in an image by mapping these pixels into a parameter space (the Hough space) defined in such a way that collinear sets of pixels in the image give rise to peaks in the Hough space.

A brief discussion of the method follows. The edge pixels processed by the transform are obtained by applying, for example, a Sobel gradient operator to the image. Such operators compute digital approximations to the gray level gradient magnitude and direction at each (non-border) pixel of the image.

Now, given a set of collinear edge points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, we know that they all must satisfy the *normal-angle* representation of a line:

$$\rho = x_i \times \cos\theta + y_i \times \sin\theta$$

The key observation is that points lying on the same straight line in the picture plane correspond to curves through a common point in the ρ, θ parameter plane. Thus, the problem of finding the set of lines in the image plane is reduced to that of finding common points of intersection of sinusoidal curves in the parameter plane.

The implementation of the Hough transform for detecting straight lines on a sequential machine involves a quantization of the parameter plane into a quadruled grid. The grid size is determined by the acceptable errors in the parameter values and the quantization is confined to a specific region of the parameter plane determined by the range of parameter values. A two-dimensional array (the accumulator array or the Hough array) is then used to represent the parameter plane grid, where each array entry corresponds to a grid cell. For each edge pixel, the algorithm increments the counts in all accumulator array entries that correspond to lines passing through that edge pixel. The process of incrementing accumulator array counts can be thought of as "voting" by the edge pixels for the parameter values of possible lines passing through these points. The time required to execute this algorithm is proportional to the size s of the grid, plus the number m of edge pixels times the number of votes v cast by each point i.e. O $(s + mv)$. The memory space is proportional to the size of the grid.

Once the Hough array is computed, we seek to find those array elements which have high values. (The number of these depends upon the constraints posed by the problem, but it is clear that we are going to seek a subset of the total available entries.) However, it is not just enough to pick the first few high values because, depending upon the quantization, the same edge pixel will contribute to a "thick" curve, i.e. votes to entries that are adjacent. More importantly, because of the inherent non-linearity of the transformation, a "real" cell will be surrounded by "false" votes in its neighborhood. These problems are solved by searching for the local maximum in an $I \times I$ square neighborhood, perhaps after smoothing the Hough array.

The Hough transform has an extensive literature. In the sequential case, Hough originally proposed a slope-intercept parameterization. Duda suggested the normal-angle representation of a straight line in order to bound the values of the parameters. The technique has been extended by Kimme et al. [KBS75] to detect other curves defined analytically, and by Ballard [Bal81] to detect general curve shapes using the edge orientations at the image points. Yet another variation which avoids the use of transcendental functions is presented in [Wal85]. Brown [Bro84] discusses a memory efficient implementation of the Hough transform.

Continued interest has also been shown in the parallel version. Silberberg [Sil85] describes a Hough algorithm on a mesh-connected computer. This has been subsequently improved upon by Rosenfeld et al. [ROH86]. They consider multiprocessors executing instructions in a Single Instruction Multiple Data (SIMD) [Fly72] fashion and where the amount of local memory available is critical. Merlin et al. [Mer75] have a method for detecting a general curve in parallel and Ibrahim et al. [IKS85] have discussed a simulation on the NON-VON [Rab86] architecture; they present two algorithms based on an SIMD and an MSIMD approach on a tree-like machine. Sher [She85] has described an implementation of the general template matching problem on the WARP systolic multiprocessor and on a Butterfly. His implementation is very different from the one described here, as discussed later.

In the simplest case, the steps involved in detecting large collinear sets of edge or feature points in a binary picture may be summarized as follows: Let θ_{min} to θ_{max} be the range of angles that we are interested in and let *Accum* refer to the accumulator array. Then the simple algorithm is :

for each edge point (x, y)

for $\theta = \theta_{\text{min}}, \theta_{\text{max}}, \Delta\theta,$

begin

$\rho = \lceil x \times \cos\theta + y \times \sin\theta \rceil$

Accum $[\rho, \theta]$ = Accum $[\rho, \theta] + 1$

end

Once the accumulator array is computed, we must next find the k highest local maxima. One can then construct another array which has as its entries all those elements which are the local maxima in all 3×3 neighborhoods. Next we could find the first k values (for instance by sorting the newly constructed array). In the sequential case, there is an obvious better algorithm which skips the intermediate step and directly finds the first k peaks. This is done using an array of k items (called final_list) which stores the angle, distance and Hough entry values. The corresponding simple algorithm that suggests itself is

for each (ρ, θ) pair entry

    if local_maxima(entry) then

    insert(angle, distance, value) in final_list.

It is not obvious if this algorithm will work better in the parallel case and we shall have more to say about this in Section 4.

3 Description of the Machines

In this section, we review those aspects of the parallel machines that are relevant to the problem at hand. For more information, see [The86,Chr86,NCU85].

The NCUBE is a "message-passing" machine configured in the form of a 10-dimensional hypercube. Advantages of this topology have been discussed in [Sei85,Pea77] etc. Each

of the 1024 identical custom built processors has "local" memory of 128K bytes and is connected to 10 other processors. The processor in the NCUBE is a general purpose 32 bit processor and includes high speed 32 and 64 bit IEEE floating point.

It is important to view the NCUBE in relation to a host Intel 286 minicomputer because code may be loaded onto the processors (interchangeably called "nodes") only from the host. Parallelism in the NCUBE is very natural after the host ships out (perhaps different) programs to the different processors. They start working on the data (provided by the host or some other node) and synchronization is achieved by sending messages. At the current stage, the main mechanism for programming is a modified FORTRAN language. Parallelism will be reduced whenever the processors are idle, perhaps due to improper load balancing, and when messages are received, sent or forwarded by the processors.

The Butterfly is a tightly coupled "shared-memory" architecture machine with at most 256 MC68000 processors. Each processor sits on a card with a process node controller (PNC) and a megabyte of memory (which in some sense is local to this processor). Processors are interconnected in a "butterfly" [KS84] fashion, thus enabling processors to gain quick access to memory located physically on a different card. There is a single clock in the whole machine but asynchronous multiple instructions may be executed by the different processors on multiple data.

The standard approach to programming the Butterfly is the so-called Uniform System [Cro86] wherein computational tasks that comprise an application are emphasized and the notion of processes is less relevant. The Uniform system views the conglomeration of memory as one huge shared memory seen in the address space of all identical processors and seeks to scatter application data to reduce memory contention. Parallelism is exploited by generating tasks using a "generator" (similar to the map function in LISP). This is typically visualized as a procedure which has as its arguments a description of the data and a worker function to analyze the data; the generator results in activating processors with

different subsets of the data. The form of this subset is generally limited in the current release (version 2.2.3) and typically is a row of a matrix.

Parallelism in the Butterfly suffers because of memory contention, switch contention and overhead involved in generating the tasks. Unfortunately, in spite of the existence of a scheduler, it is not possible to generate more active processes than processors in the Uniform system. This has imposed a limitation on our experiments. Nor is it possible for processors to act "independently" since the same code is loaded on all the processors.

4 The Algorithms

We present our parallel algorithms in this Section. As mentioned earlier, one has to compute the Hough array and then extract peaks. On a systolic machine, such as the Warp, one may be interested in detecting the peaks even while the Hough array is being calculated, but on our machines there is no obvious advantage to this approach. We construct the Hough array and detect the peaks in a sequential manner but in both the steps we would like to allocate processors optimally.

4.1 Computing the Hough array

Let $\theta_{\min}$ to $\theta_{\max}$ determine the relevant range of angles. One may think of computing the Hough array in terms of mapping a three-dimensional $x \times y \times \theta$ space into a one-dimensional ρ space and then incrementing the appropriate (ρ, θ) cells by 1.

At this stage, it is not obvious how the processors should be allocated. For instance, we can identify the following options:

- Distribute processing by x: Assign one row of the binary image to each processor. It then computes ρ over the entire range of angles. (One could conceivably use a column instead of a row.)

- Distribute processing by region: Assign a square shaped region of the input binary image to a processor. It then computes ρ over the entire range of angles.
- Distribute processing by θ: Assign one angle to each processor. It then computes ρ for the entire binary image.
- Distribute processing by θ: As above, assign a processor to an angle. However, the processor analyzes only a part of the input image.

As an example, consider scaling the problem down to an input picture of size 1×2, an angle spread of 5° and an angular quantization of 1° (an assumption we will make throughout, for simplicity). Assuming that we have only two processors, we immediately see that it is preferable to distribute processing by θ. The only thing that we can hope for is that the input image is random. If we allocate one processor to each of the pixels, then one processor may end up doing no work at all since it may not be the case that both the pixels are edge pixels. On the other hand, distributing by θ implies that both processors are working at all times and will, in general, lead to better processor allocation. This corresponds to the case of "Distribute processing by θ" above and will be the pivotal idea on which we shall base the allocation upon.

We present an allocation scheme for the general case (other schemes are also possible). Let $\Psi \equiv \theta_{\max} - \theta_{\min} + 1$ and let N be the number of rows in the input image. We will consider the following three possible cases:

$$P < \Psi \tag{1}$$

$$\Psi < P < \Psi N \tag{2}$$

$$\Psi N < P \tag{3}$$

In the first case, the ith processor computes the Hough array for a set of θs; specifically, $\theta_{\min} + i, \theta_{\min} + i + P, \ldots, \theta_{\min} + i + \lceil \Psi(modulo P) \rceil$ [3] over the entire binary image.

In the second case, the unit of processing given to a processor is a row of the input image and an angle. For each edge point in the row, the processor computes (or looks up in a table) the appropriate value of ρ. Each processor can be given either overlapping sets of rows or "scattered" sets (our implementation).

Finally, in the last case, the unit of processing is now an angle and part of the input row. Thus, for instance, if $P = 2\Psi N$, then processors will be working on an angle and half a row of the input image.

4.1.1 Computational complexity for the Hough array computation

The analysis for this part is easy; it illustrates the relation between the problem size and the efficiency of the parallel computation.

Once the input space is allocated to the different processors as described above, each processor independently computes a "version" of the Hough array based on the subset of the problem given to it. The Hough arrays are then collapsed into the complete Hough array using the standard [PV81] collapsing technique. Thus, on the hypercube, one could use the following procedure, Algorithm Merge, to merge the Hough arrays (here, *mynodeid* is a unique address for a hypercube node):

for $index = 1$ to $\log_2 P$

begin

if mynodeid $\leq 2^{index}$ then

neighbor = *mynodeid* xor $2^{index-1}$

if neighbor < *mynodeid* then

send (myhougharray, neighbor)

[3] A different partioning where each processor analyzes contiguous angles is possible.

```
else
    begin
    receive (hishougharray, neighbor)
    myhougharray = myhougharray + hishougharray
    end
end
```

Let H_size represent the size of the Hough array. For the purpose of analysis we assume that every pixel in the input $N \times N$ image is an edge pixel in the worst case and that H_size is asymptotically smaller than the input N. Then, in this model of computation,

$$T_P(N) = O(N^2/P) + H\_size \log P \tag{4}$$

which we shall approximate as

$$T_P(N) = O(N^2/P). \tag{5}$$

Recall that

$$T_1(N) = O(N^2). \tag{6}$$

Munro and Patterson [MP71] have a lower bound result which states that

$$T_P(N) \geq T_1(N)/P + \log_2(\min(P, T_1(N))) - 1. \tag{7}$$

If we substitute equation 6 into equation 7, we have

$$T_P(N) \geq O(N^2/P) + \log_2(\min(P, N^2)) - 1 \tag{8}$$

and thus asymptotically, this result is optimal within the constraints of our assumption. (See eqn. 5.)

As discussed in the introduction, it is useful to see for what values of input size this algorithm achieves optimal speedup. Essentially we want to determine when

$$T_P(N) = T_1(N)/P$$

viewing P and N in turn as the independent variable. We thus need to solve the equation

$$N^2/P + H\_size \log_2 P = O(N^2/P)$$

The solution to this is

$$N = \Omega(H\_size P \log P)^{1/2}.$$

The problem size should be at least $(P \log P)^{1/2}$ for the algorithm to be optimal or the DEF is $(P \log P)^{1/2}$.

Reversing the question we have, for optimality,

$$P = N^2 / \log N$$

since the term

$$H\_size \log_2 P$$

evaluates to

$$2H\_size \log(N/\log N)$$

or

$$O(\log N)$$

In other words, the algorithm exploits at most $N^2/\log N$ processors to solve a problem of size N and this gives an idea of how good the algorithm is (the PEF is $N^2/\log N$). To summarize, the algorithm is in the Class $J_{(P \log P)^{1/2}}$ and Class $K_{N^2/\log N}$.

4.1.2 Implementation details

Recall from the description in Section 3 that on the Butterfly we do not have the ability to increase the number of processes beyond the actual number of physical processors, i.e. one cannot have virtual processors. This limits the set of experiments that we can perform since we cannot measure the speedup beyond 16 processors.

We remarked earlier that there are other data allocation schemes that may also achieve similar performance. Our scheme has the additional merit that on the Butterfly it is easy to implement. When the Uniform System generator produces tasks to be executed in parallel, it produces, conceptually, an *identification* number for each task and different processors execute tasks corresponding to different identification numbers. In our situation, when we allocated angles to each processor (corresponding to Equation 1 in Section 4.1), these numbers correspond to the different angles since now the unit of processing is an angle. As a result, the P processors evaluate the first P tasks in parallel and then process the next P tasks. (Each processor thereby does not work on the task corresponding to contiguous angles.) In the next situation (see Equation 2 in Section 4.1) we decided to allocate rows of the input image for a similar reason and thus each processor does not work on contiguous rows but rather on a scattered set of rows. It also helps that adjacent elements in a row of the matrix are also physically adjacent even though adjacent rows are scattered since *block* transfer of physically adjacent memory cells from global memory to the local memory of the processor is efficient on the Butterfly. Thus there is no contention when different processors are accessing the input image in parallel.

Once the individual Hough array versions are computed by the different processors, we need to merge them into one "correct" Hough array. Communication between processors on the shared memory Butterfly is achieved by directly accessing global memory, so in Algorithm Merge, we need not explicitly send or receive the individual Hough array versions to be merged. However, the "versions" that the different processors compute have to be allocated in global memory and there is an obvious duplication here; there are bound to be situations where this might be a limitation since, in particular, the current Uniform System forbids piecewise freeing of dynamic memory.

Our implementation defined the "final" Hough array in global memory. The individual versions are never explicitly built but instead the copy in global memory is updated. It

is well known that updating in global memory requires that the information be added atomically; so we use the "locking" mechanisms available in Chrysalis to ensure integrity. Thus, there is no need to allocate memory in the global address space for the individual versions.

Undoubtedly this approach reduces efficiency; however, one can manipulate the program variables to take advantage of the local and global memory aspects of the shared memory and reduce the contentions. Since programming on the Butterfly is done in C, we often need to follow a sequence of pointers in order to access the "locked" data structure. A useful strategy in such cases is to make local copies of the data thus restricting contention only at the final destination memory word. In fact, for the Hough transform done on a window of an image for the Autonomous Land Vehicle, we observed linear speedup for a 32×32 input binary image with approximately twenty-five percent edge pixels when the angle spread was about twenty degrees.

The same idea may be used even in the processor allocation step. In addition, accesses to local memory cost less than accesses to global memory; so a "block" of data may be transferred to local memory and then used in computations. Again, when processors are allocated parts of the input matrix, allocation is done *alternatingly from the top and the bottom.* This results in less contention while accessing common data, for instance, overlapping rows.

Though the algorithm suited the NCUBE very well, it was in its infancy of software development when the algorithm was implemented. For instance, the time spent in execution of the program viz. the duration between sending of the node program and receiving the information back to the host (recall the discussion in Section 3) cannot be easily be measured. Secondly, it is possible to pick messages out of order in which they have been sent, i.e. by the type of message. Finally, input/output is possible only from the host and this restricts the pace of debugging programs.

4.2 Peak detection

More interesting theoretical ideas are involved in searching for the k highest peaks once the accumulator array is available and is in fact the subject of another report [CR86]. We propose different solutions here and consider their relative advantages.

Using a single processor, the problem cannot be solved faster than $O(N)$ because the kth largest value could be the Nth value (here N refers to the size of the Hough array and not the image size). Recall further that one can find the median of a list of numbers (or in general, the kth largest element) in linear time [AHU74]. Once we do this, we can look at one element at a time, compare it to the kth largest element and thus find all the desired elements in linear time. Thus, although we search for the k highest peaks, our problem is in fact no more difficult than the median finding problem. Our goal is to solve the problem in constant time using N processors and, in general, to solve it in N/P time using P processors.

At first, one is inclined to sort the Hough bins using the accumulator value as the key. (Once sorting is done, it is trivial to pick the k highest peaks.) Using N processors, we cannot sort in less than $O(\log N)$ time [AKS81]. As far as implementations go, though the above is fast, it is not practical because the constant hidden in front of the $O(\log N)$ description is very large (about a million). The value of k in most applications is low and the following algorithm is an efficient alternative (though asymptotically not faster than sorting).

Processor allocation is done by splitting (but not partitioning) the Hough array into P strips. When successive strips share a common row, then each pixel and its 3×3 neighborhood is contained in some single strip (except for the border pixels). Then each processor using a fast sorting algorithm finds the first k elements in its data. We then combine the individual k peaks in a tree-like fashion to produce the final desired k peaks.

This algorithm may be easily implemented both on the Butterfly and on the NCUBE. It is easy to observe that

$$T_P(N) = O(N/P \log(N/P) + k \log P),$$

the first term arising from the sorting part and the second from the merging. Asymptotically, this is not a good algorithm but for "reasonable" values of k and N, it is satisfactory. In fact, this is the version we have implemented.

If we are indeed interested in asymptotics, we observe that we can use the linear median finding algorithm instead of the fast sorting algorithm in the initial stage (i.e. when each processor finds the k highest peaks in its data). The merging of these is done as before. In this case,

$$T_P(N) = O(N/P + k \log P). \tag{9}$$

Once again, we are interested in when, as a function of N and P, this algorithm achieves a speedup of P over the sequential linear running time. If the problem size is at least $kP \log P$ optimal efficiency is achieved (the DEF is $kP \log P$) [4]. Once again, reversing the question, if we have at most $N/(k \log N)$ processors, then substituting into equation 9, we have

$$T_P(N) = O(k \log N + k \log(N/k \log N))$$

i.e.

$$T_P(N) = O(k \log N)$$

and thus

$$PT_P(N) = O(N)$$

the uniprocessor time. The problem is solved optimally and to summarize, the algorithm is in the Class $J_{(kP \log P)}$ and Class $K_{N/(k \log N)}$.

[4] Of course, here we are assuming that k is not a parameter to the problem; [CR86] describes an algorithm whose running time is independent of k.

5 Conclusions

The Hough transform is a technique to detect line segments in an image by finding global consistencies in the data. Since it considers every feature point, the process is slow on a conventional Von Neumann machine. There is, however, a certain independence in the Hough array computation and whenever such independence exists in any problem, the use of machines whose architecture is not Von Neumann is justified. Matrix multiplication is an example of a problem wherein all the computations are independent and is an excellent test problem for a parallel machine. An appropriate partitioning of the data is required in such problems and in this paper, we have identified the "independent" component in Section 4.1.1 and shown how to allocate processors on the NCUBE and the Butterfly. Once the Hough array is computed, one is usually interested in detecting peaks in order to find the best line segments. We have shown that this problem is no more difficult than the median finding problem and proceeded to obtain an optimal parallel selection algorithm. Here the partitioning of data is less critical and the communication costs dominate the efficiency of the parallel computation.

An orthogonal issue that we have addressed in this paper is the relation between problem size and the number of processors. Given a machine of a fixed configuration, it is important to find the appropriate problem size to determine the efficacy of the architecture; in most problems, if the problem size is sufficiently large, one is bound to get a linear speedup on the finite number of processors that exist on the machine. We are interested in finding bounds on this parameter. Alternatively, when one has a fixed application, and the problem size is constrained by other issues, one is interested in determining a reasonable number of processors that one should use for the problem. It is clear that not all the processors may be useful if, for instance, the problem is too small. By introducing the idea of processor and data efficiency functions, we have illustrated a technique to obtain

bounds on these numbers.

Acknowledgements

We thank Deepak Sherlekar for insights into the parallel selection problem.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading Massachusetts, 1974.

[AI86] Arvind and R. Iannucci. *Two Fundamental Issues in Multiprocessing.* Technical Report, Massachusetts Institute of Technology, 1986.

[AKS81] M. Ajtai, J. Komlos, and E. Szemeredi. An $n \log n$ sorting network. *Combinatorics*, 3(1), 1981.

[Bal81] D. Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2), 1981.

[Bro84] C. Brown. Peak finding with limited hierarchical memory. In *7th International Conference on Pattern Recognition*, 1984.

[Chr86] *Chrysalis Programmers Manual, Version 2.2.3.* Bolt Beranek Newman, 1986.

[CR86] S. Chandran and A. Rosenfeld. Selection on the NCUBE. Manuscript in preparation, 1986. Center for Automation Research.

[Cro86] W. Crowther. *Using the Uniform System to program the Butterfly.* Bolt Beranek and Newman Inc., 1986.

[DH72] R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1), 1972.

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9), 1972.

[Hou62] P. V. C. Hough. Method and means for recognizing complex patterns. U.S. Patent 3,069,654, 1962.

[IKS85] H. Ibrahim, J. Kender, and D. Shaw. *The Analysis and Performance of Two Middle-Level Vision Tasks on a Fine-Grained SIMD Tree Machine.* Technical Report, Columbia University, 1985.

[KBS75] C. Kimme, D. Ballard, and J. Sklansky. Finding circles by an array of accumulators. *Communications of the ACM*, 18(2), 1975.

[KS84] C. Kruskal and M. Snir. Optimal interconnection network. In *11th Annual International Symposium on Computer Architecture*, 1984.

[Mer75] A. Merlin. A parallel mechanism for detecting curves in pictures. *IEEE Transactions on Computers*, 24(1), 1975.

[MP71] I. Munro and M. Paterson. Optimal algorithms for parallel polynomial evaluation. In *12th Annual Symposium on Switching and Automata Theory*, 1971. Also see Journal of System Sciences,7, 1973.

[NCU85] *NCUBE Handbook, Version 0.6.* NCUBE Corporation, 1985.

[Pea77] M. Pease. The indirect binary n-cube microprocessor array. *IEEE Transactions on Computers*, 26(5), 1977.

[PV81] F. Preparata and J. Vuillemin. The cube-connected-cycle. *Communications of the ACM*, 24(5), 1981.

[Rab86] G. Rabbat. *Computers and Technology.* Elsevier-North Holland, 1986.

[ROH86] A. Rosenfeld, J. Ornelas, and Y. Hung. *Hough Transform Algorithms for Mesh-Connected SIMD Parallel Processors.* Technical Report CAR-TR-128, Center for Automation Research, 1986.

[Ros69] A. Rosenfeld. *Picture Processing by Computer.* Academic Press, New York, 1969.

[Sei85] C. Seitz. The cosmic cube. *Communications of the ACM*, 28(1), 1985.

[She85] D. Sher. *Template matching in parallel.* Technical Report, University of Rochester, 1985.

[Sil85] T. Silberberg. The Hough transform on the Geometric Arithmetic Parallel Processor. In *Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Mangement*, 1985.

[The86] *The Butterfly Parallel Processor Overview.* Bolt Beranek and Newman Inc, 1986.

[Wal85] R. Wallace. A modified Hough transform for lines. In *Computer Vision and Pattern Recognition*, 1985.

END

4-87

DTIC