Abstract - This paper presents algorithms for Hough Transform on Mesh of Trees parallel computers. Using a Mesh of Trees with \( n \times n \) processors on the base, our algorithms perform Hough Transform for an image of size \( n \times n \) in \( O(n) \) time, with \( O(n) \), \( O(\sqrt{n}) \), and \( O(n(\log n)) \) memory usage, respectively. The algorithms were simulated, and the results were compared with those of existing algorithms. It is shown that our algorithms are efficient, and can achieve a good speedup in real-time Hough Transform computation.

keywords: algorithm, image processing, Hough Transform, Mesh of Trees, complexity.

INTRODUCTION

The Hough Transform (HT) has been found very useful in computer vision and is frequently used in detecting the shape of object boundaries in image pattern analysis [2, 3]. It is to detect the presence of groups of collinear or almost collinear pixels. The amount of computation required in the HT grows as the number of pixels increases and the demanded accuracy (known as resolution raises). Because the amount of computation is very large, it is not easily implementable in real-time calculation. In recent years, a number of approaches for parallel implementation of the HT have been reported. Assume that the size of the parameter plane is \( n \times n \) and the number of the pixels is \( N \). A HT algorithm with time complexity at least \( O(N^2) \) on a fine-grain SIMD machine with broadcasting capability has been proposed by Ibrahim [4]. Silberberg has reported an \( O(n^2) \) parallel method in GAPP (Geometric Arithmetic Parallel Processor) [5]. The systolic array for an improved HT which was reported by Chuang et al. requires \( O(N) \) time [6]. An algorithm that has been described on mesh arrays takes \( O(N + n) \) time and \( O(n) \) memory [7].

Hough Transform

Basically, HT involves transforming each of the edge pixels in an image plane to a parameter plane. A straight line in the \((x, y)\) image plane can be described by \((a, b)\) where \(a\) and \(b\) represent slope and intercept respectively and \(y = ax + b\) is satisfied. On the other side, it can be parameterized by \((p, \theta)\) where \(p\) is the distance between the origin and the line; \(\theta\) is the angle between the line and the positive \(x\) axis measured counterclockwise, and

\[
p = x \cos \theta + y \sin \theta
\]

is satisfied. The most commonly used coordinate system in the HT implementation is \((p, \theta)\).

Given a set of \( n \) pixels, \([(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]\), it is possible to find a set of straight lines that connect some of these points together. A pixel at location \((x, y)\) is transformed into the sinusoidal curve in the \((p, \theta)\) plane defined by equation (1) by varying \(\theta\). Curves corresponding to the set of collinear figure points in the \((x, y)\) plane will have a common point. Thus the problem of detecting collinear points can be converted to the problem of finding concurrent curves. (See Fig. 1) When it is not necessary to determine the lines exactly, we can specify the acceptable error resolution \(\delta_{\text{err}}\) and \(\rho_{\text{err}}\), and quantize \(\theta\) into \(n\) values \(\theta_0, \ldots, \theta_{n-1}\), and \(p\) into \(n\) values \(p_0, \ldots, p_{n-1}\). To keep track of the count of edge pixels, we keep the record in a 2-dimensional array \(H\) in which each entry corresponds to a point defined by a pair \((p, \theta)\). Now, equation (1) is rewritten as:

\[
p = (x \cos \theta + y \sin \theta) / \rho_{\text{err}}.
\]

For each point \((x, y)\) in the image plane, the \((p, \theta)\) cell defined by equation (1) is incremented. A given cell \(H\) counts the total number of curves intersecting at the point represented by a given \((p, \theta)\) in the parameter plane. After all the pixels have been treated, \(H\) is inspected to find those cells with the largest counts. If the count value in a given cell \((p, \theta)\) is \(n\), then precisely \(n\) edge pixels lie (within the quantization error) along the line which is defined by \((p, \theta)\).

Assume that the image contains \(n^2\) pixels and that the size of the parameter space is \(n \times n\), i.e., the range of \(\theta\) and that of \(p\) are quantized into \(n\) values \(\theta_0, \ldots, \theta_{n-1}\), and \(p_0, \ldots, p_{n-1}\), respectively. Obviously, the time complexity for the sequential calculation is \(O(n^3)\). As the image and the number of quantized values along the \(\theta\) direction can be very large, faster methods have to be developed for real-time applications. One approach to speed up the calculation is to use parallel processors to perform the intensive computation in parallel. A number of results have been reported [4, 5, 7, 6]. Searching for parallel architecture to accelerate our computation, we identified the Mesh of Trees as a promising computer for performing HT.

MESH OF TREES

The Mesh of Trees computer[1] is constructed from an \(n \times n\) matrix of processors by adding processors and wires to form a complete binary tree in each row and each column. For the problems considered in this paper, the roots (numbered from 0 to \(n - 1\)) of the trees are used for input/output. Most of the processing is done by the base processors (BEs). The internal processors (IE's) usually are used for communication between BEs or between IE's. During the course of this communication the IE's may also be required to carry out some simple operations such as summing, comparison and extracting the maximum data. As an example, a \(4 \times 4\) Mesh of Trees is shown in Fig. 2. Following are the commonly used communication operations used in this paper.

- **ROOTOLEAF** (Vector, Obj): “Vector” refers to either rows or columns of BEs. “Obj” refers to a corresponding selected register in these BEs which will receive the broadcast data.

---

Figure 1: Representation of points in polar coordinates.

Figure 2: The two-dimension Mesh of Trees (\(4 \times 4\)).
• LEAFFTORoot(Vector, Source): "Source" has the same meaning as the Obj except that it will send out data.

• LEAFSTOSUBRoot(Vector, Source): has the same meaning as the LEAFFTORoot(Vector, Source) except that the SUBRoot refers to a root of a subtree which will be defined later.

• IETORoot(Vector, Source): "Vector" refers to either all or part of IE's in a subtree. "Source" refers to a corresponding selected register in the root IE's which will send out the data.

A GROUP OF ALGORITHMS

We assume that the image size is $n \times n$ and the parameter space is also $n \times n$. The main strategy to design our algorithms is: each BE in $j$th(where $0 \leq j \leq n$) column only owns a $\theta$ value, $\theta_j$, and a one-dimensional array represented by $h_j$ does the count work. The image pixels are input into the row of the root and are passed down through the row tree to BEs. BEs calculate the $p$ value by applying equation 2 one pixel at a time and accumulate the corresponding entry in the $h_j$. After all pixels are processed, the data in the entries of $h_j$ is passed up one by one through the column tree to the root of columns where the maximum value in all entries of $h_j$ will be found.

The First Algorithm

Broadcast. $\sin \theta_j$ and $\cos \theta_j$ are input into the root of the $j$th column for $0 \leq j \leq n-1$ and are consecutively passed in a pipeline fashion downward through the column trees. $h_j$ (where $j$ is the index of column) is reserved in each BE.

Procedure Broadcasting

for each $j(0 \leq j \leq n-1)$

1. Pardo

2. ROOTOLEAF (column, j, h) ($\sin \theta_j$);

3. ROOTOLEAF (column, j, h) ($\cos \theta_j$);

end for

In the above procedure, pardo indicates that Procedure Broadcasting is performed in a pipeline fashion assuming $\sin \theta_j$ or $\cos \theta_j$ to be available successively at the row input ports.

Calculation. First, the pixels are entered into the root of the row. For an image containing $n \times n$ pixels, the $i$th row pixels, namely, $u$ pixels, are input into the root of the $i$th row in a pipeline fashion. In other words, each root of the row takes care of one row pixels. Simultaneously, the value of the pixel is passed down via the tree one by one so that the value of the first pixel arrives at each BE of corresponding row in log $n$ steps. BE($i, j$) can now compute the $p$ value by applying equation 2 and increments the $p$th element of the $h_j$ array. Then, BE($i, j$) can accept the next pixel. This process is repeated $n$ times.

However, we do not perform calculation in the above way and we can have some modifications to save calculation time. Two multiplications and one division are required in each cycle when we calculate the HT by applying equation 2 in the above algorithm. They are all expensive floating point operations and are time-consuming. So, it would be advantageous if we can replace these operations by less expensive and less time-consuming additions or subtractions. In [7], Kannan introduced a method to realize this goal. His idea can be similarly applied to our algorithm. The modified algorithm is given below.

Procedure Calculation

Input pixel values of $i$th row to the root of the $i$th row successively and pass down through the row tree in a pipeline fashion;

1. In each BE

2. $p(u) = \cos \theta_j / \sqrt{\rho}$; $\rho = j \cdot \sin \theta_j / \rho$;

3. if pixel 1 = 1 then $h[k] = h[k] + 1$;

4. accept the 2nd pixel from the parent of BE in the row tree;

5. for $k = 1$ to $k = n-1$ do

6. $p = p + \rho$;

7. if pixel 1 = 2 then $h[k] = h[k] + 1$;

8. accept the next pixel from the parent of BE in the row tree;

end for

Summation. Because the same value of $\theta$ is held in all processors of each column of BEs, we have accumulated the counts for $(\rho, \theta_j)$, $0 \leq i, j \leq n-1$, in the $h_j$ array of the BE on the $j$th column. The contents of the corresponding elements of the $h_j$ array on a column of BEs should be summed up to get the total counts, and this should be done in all $n$ columns in parallel. All BEs send the contents of the first cell, namely, $h_j(0)$, to their parents in the column tree. The parents of BEs then add the two values received from their two children and send the result to their parents in the same column tree (i.e., the grandparent of BEs). Simultaneously, the BEs send the contents of the second cell, $h_j(1)$ to their parents. The similar tasks are executed in the subsequent cycles. At last, the root of the $j$th column will have the final value of $h_j$ (corresponding to $\theta_j$).

Procedure Summation

for $k = 0$ to $n-1$ pardo

1. Send $h_j(k)$ to the $C$ register of its own BE;

2. LEAFFTORoot (column all, C);

end for

Max. After all the roots of columns have received $h_j$ for the corresponding $\theta$ at the end of previous phase, the largest value is to be found in the $H$ array of the root of each column.

Line. One more step is needed here to find the largest value in the 2-dimensional $H$ array, i.e., the maximum for all the $\theta$'s.

Procedure FindMaximum

1. ROOTOLEAF (column all, A(row 0));

2. LEAFFTORoot (row 0, A);

Time Complexity. The time complexity of the first algorithm is $O(n^2)$, and the memory space is $O(n)$.

The Second Algorithm

With $O(n^3)$ sequential computation of HT and $O(n^2)$ processors of the Mesh of Trees is, the first algorithm achieves an optimal time complexity $O(n)$. However, the memory usage can be further reduced.

The straight line we want to detect can be specified by $\rho$ and $\theta$ where $\rho$ is the distance from the origin to the line. Therefore, the maximum $\rho$ must be the diagonal of the image, $\sqrt{n^2+n^2}=\sqrt{2n^2}$ for an image of size $n \times n$ (in pixel units). In the first algorithm, each BE has to process $n$ pixels with range of horizontal coordinate from 0 to $n-1$ and vertical coordinate remaining unchanged. The changeable range of $\rho$ for each processor is $\sqrt{n^2}$, and each BE has to store a $p$-list of length $n$. In the following algorithm the length of $p$-list for each BE is reduced, and the memory usage hence decreases.

Instead of that each BE processes one row pixels (of number $n$), we let it process $\sqrt{n} \times \sqrt{n}$ pixels (i.e., a $\sqrt{n} \times \sqrt{n}$ square of the image). The number of pixels for each BE to process remains unchanged. Because each BE processes one $\theta$, the largest changeable range of $\rho$ with respect to each $\theta$ is from the first pixel to the last pixel in the small $\sqrt{n} \times \sqrt{n}$ square. Obviously, it is $\sqrt{n}$ if we do not consider the resolution. The distribution of the image for this algorithm is shown in Fig. 3.

![Figure 3: Distribution of image pixels for the second algorithm.](image-url)
Training. By broadcasting sin θ and cos θ to the corresponding BE first, the similar method used in the first algorithm can be modified and used to calculate the p-list.

Procedure Training
for each (0 ≤ i ≤ n − 1) pardo
ROOTOFFSET (column c, A) (sin θ);
ROOTOFFSET (column c, B) (cos θ);
end for

Procedure Calculation
Let pd(a) be the index of the root of the row;
In the root of each row
pid = pd(a) mod √n;
start_x = pid * π / √n, end_x = (pid + 1) * π / √n − 1;
start_y = (pd(a) / √n) * √n, end_y = ((pd(a) / √n) + 1) * √n − 1;
for i = start y to √n end_y do
if 0 =< i < end_y do
p = cos (b) * sin θ + p;
else
p = p + p;
end if
end for

Calculation. Input the values of each small √n × √n matrix into the corresponding root processor element of each row. These values are passed downward through the row tree in a pipeline fashion provided that the values are available successively at the row input ports. After log n steps, each BE receives the first value. Now, we can perform the calculation.

Procedure Calculation
Let pd(a) be the index of the root of the row;
In the root of each row
pid = pd(a) mod √n;
start_x = pid * π / √n, end_x = (pid + 1) * π / √n − 1;
start_y = (pd(a) / √n) * √n, end_y = ((pd(a) / √n) + 1) * √n − 1;
for i = start y to √n end_y do
if 0 =< i < end_y do
p = cos (b) * sin θ + p;
else
p = p + p;
end if
end for

Summation. Sum up the contents of the corresponding elements of the h_i array on a column of BE to obtain the total counts. The method is similar to the Summation phase in the first algorithm, except that the processors are needed to compare the current received value with previously received ones, to choose the bigger one and to save it for the next comparison. After n cycles, the largest value for each θ can be obtained. In the following algorithm, we do not consider the comparison performed in root element, because it only needs one step for each cycle and it will not affect the time complexity.

Procedure Summation
Each BE pardo
flag = 0;
end if
if (flag == 1) then
LEAEHTOROOT (column all, h_j[p]);
end if
end if
end for
Having the largest θ value for each θ, we then call the subroutine FindMaximum in Algorithm 1 to find the maximum value for all the θ's.

Time Complexity. The time complexity of the second algorithm is O(n) and the space complexity is O(√n).

The Third Algorithm
Another algorithm is introduced below which does not need the training phase and hence has a smaller factor in the same O(n) time complexity. Moreover, The memory complexity is also less than O(n).

In this algorithm, each BE processes one row pixels which are divided into log n groups, and each group is with n/log n pixels. We distribute uniformly the p list to the processors in the lower half of the column tree, with (2 log n/n) entries per processor.

Initially, each row BEs calculate the p value for the first group of (n/log n) pixels. Depending on the p value, a corresponding entry will be found in the temporary array maintained by the BE, and the count for it will be incremented. After the first group is processed, the count of the ith entry for p, in the temporary array will be passed to the root of the column tree. On its way, one of the BEs will have the ith entry with the same p value, and the count will be accumulated there. We will then calculate the second group of (n/log n) pixels, performing the summation again and so on. In this way, the calculation and summation are done one group pixels after another. As we know, the lower half of the column tree contains √n subtrees, each of them having √n leaves and √n−1 entries, i.e., log √n height. Each BE in the same level stores the same part of the p-list, referred to as a "sub-list".

Theorem 1. In any subtree, after the calculation of any group pixels, if we request that all the BEs send out the resulted values which are corresponding to a specified entry of all the sublists, there are totally log n entries along any path in a subtree. However, all BEs in a subtree will send out the same value which is corresponding to that specified entry in some IE and only one IE.

Proof. The range of the resulted value of a group pixels for the first BE in a subtree is (2n cos θ + y sin θ, (2n cos θ + y sin θ) + (2n cos θ + y sin θ)). The range of the resulted value of a group pixels for the last BE in a subtree is (2n cos θ + y sin θ, (2n cos θ + y sin θ) + (2n cos θ + y sin θ)). Therefore, it is obviously n/log n > √n. Therefore,

Proof completed
According to this theorem, only addition but no comparison is needed in the summing up phase. Now, we can present the third algorithm.

The Broadcasting and the Calculation phase are similar to those of algorithm 1, except that summing up is modified because the calculation in BEs are different now. Following are the calculation and part of the summing up which are accomplished in one phase.

**Procedure Calculation**

Input pixel values of ith row to the root of the ith row successively and pass down through the row tree in a pipeline fashion:

Each BE parado

\[ p = j \times \sin \theta / \pi \]

if pixel \( = 255 \) then \( temp[p] = temp[p] + 1 \);

accept the 2nd pixel from the BEs row parent,

\[ p = \cos \theta / \pi \]

Let \( k \) be the counter of group number

for \( k = 1 \) to \( k = \log_2 n \) do

for \( j = 0 \) to \( j = n / \log_2 n - 1 \) do

\[ p = p + p_c \]

if pixel \( = 255 \) then \( temp[p] = temp[p] + 1 \);

end for

end for

Let \( x \) be the first index of temp array in each BE;

\[ flag = 0, r = 0 \]

for \( i = 0 \) to \( j < 2 \times n / \log_2 n \) do

if \( (flag = 0 \) and \( j = x \mod (2 \times n / \log_2 n) \) )

then \( flag = 1 \);

end if

end for

We have obtained the \( h_r \). However, they are distributed in those subtrees and we need to obtain the maximum value for each \( \theta \), separately. It can be done as follows.

**Procedure Summation**

\[ \text{for } i = 1 \text{ to } i < \log_2 n - 1 \text{ do} \]

\[ \text{for } k = 0 \text{ to } 2 \times n / \log_2 n - 1 \text{ do} \]

\[ C(\text{register}) = h_r(4) \text{ in all BEs of those subtrees; } \]

\[ \text{LETOROOT}(IE, C); \]

end for

end for

\[ \text{max} = 0; \]

\[ \text{for } i = 1 \text{ to } i < \log_2 n + 1 \text{ do} \]

\[ \text{for } k = 0 \text{ to } 2 \times n / \log_2 n - 1 \text{ do} \]

\[ C(\text{register}) = h_r(4) \text{ in all BEs of those subtrees; } \]

\[ \text{LETOROOT}(IE, C); \]

end for

All the column processors of roots execute

for \( index = 0 \) to \( 2 \times n / \log_2 n - 1 \) do

if \( \text{max} < h(index) \) then \( \text{max} = h(index) \); end if

end for

end for

We have the largest \( p \) value for each \( \theta \) now. We can call the subroutine FindMaximum in Algorithm 1 to find the maximum value over all the \( \theta \).s.

**Time complexity.** The time complexity of the Third algorithm is \( O(n) \), and the memory usage is \( O(n / \log_2 n) \).

**SIMULATION**

We obtained the experimental results on \( n \)-CUBE2, simulating a Mesh of Trees parallel computer. The \( n \)-CUBE2 which we use has 64 processors. We process on it three sizes of images of \( 64 \times 64 \), \( 128 \times 128 \), and \( 256 \times 256 \) pixels. Following are the execution time and speed up for implementing the second algorithm. From table 2, we can see that the speed-up increases as the image size increases. When the size of image is approaching to infinite, the ideal speed up, 64, can be achieved for a system of 64 processors. (See figure 4 for reference.)

**Table 1: Execution time table (time unit usec)**

<table>
<thead>
<tr>
<th>n-CUBE2</th>
<th>One processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>time</td>
</tr>
<tr>
<td>( 64 \times 64 )</td>
<td>541928</td>
</tr>
<tr>
<td>( 128 \times 128 )</td>
<td>256 x 256</td>
</tr>
<tr>
<td>( 256 \times 256 )</td>
<td>135322</td>
</tr>
</tbody>
</table>

**Table 2: Speed-up table**

<table>
<thead>
<tr>
<th>size</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 64 \times 64 )</td>
<td>1.15</td>
</tr>
<tr>
<td>( 128 \times 128 )</td>
<td>5.61</td>
</tr>
<tr>
<td>( 256 \times 256 )</td>
<td>20.62</td>
</tr>
</tbody>
</table>

**Figure 4: Speed-up for different image size.**

**COMPARISON WITH OTHER WORKS**

We have presented several simple efficient parallel HT algorithms on Mesh of Trees. The algorithms take advantage of both Mesh and Tree network structures. With careful consideration of data flow and information management, we obtained a very good result both in time performance and in memory usage. Table 3 summarizes the known bounds in the literature and the results derived in this paper.

**Table 3: Comparison of HT algorithms, where \( N = n^4 \)**

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Pixel size</th>
<th>Memory/Proc.</th>
<th>Time</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ibrahim[4]</td>
<td>( O(1) )</td>
<td>( O(N) )</td>
<td>( O(N^2) )</td>
<td>Tree</td>
</tr>
<tr>
<td>Chuang[6]</td>
<td>( O(n) )</td>
<td>( O(N^2) )</td>
<td>( O(N + n) )</td>
<td>Syntonic</td>
</tr>
<tr>
<td>Silber[8]</td>
<td>( O(1) )</td>
<td>( O(N \times C) )</td>
<td>( O(N) )</td>
<td>Mesh</td>
</tr>
<tr>
<td>Kanani[1]</td>
<td>( n^2 )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>Mesh of Trees</td>
</tr>
<tr>
<td>Alg1</td>
<td>( n^2 )</td>
<td>( O(n^2) )</td>
<td>( O(n) )</td>
<td>Mesh of Trees</td>
</tr>
<tr>
<td>Alg2</td>
<td>( n^2 )</td>
<td>( C \times N )</td>
<td>( O(n) )</td>
<td>Mesh of Trees</td>
</tr>
<tr>
<td>Alg3</td>
<td>( n^2 )</td>
<td>( (n \times \log_2 n) )</td>
<td>( O(n) )</td>
<td>Mesh of Trees</td>
</tr>
</tbody>
</table>

**References**


