



# A Fast Efficient Parallel Hough Transform Algorithm on LARPBS\*

LING CHEN AND HONGJIAN CHEN

*Department of Computer Science, Yangzhou University, Yangzhou 225009, China*

YI PAN

*Department of Computer Science, Georgia State University, Atlanta, GA 30303, USA*

YIXIN CHEN

*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA*

**Abstract.** A parallel algorithm for Hough transform on a linear array with reconfigurable pipeline bus system (LARPBS) is presented. Suppose the number of  $\theta$ -values to be considered is  $m$ , for an image with  $n \times n$  pixels, the algorithm can complete Hough transform in  $O(1)$  time using  $mn^2$  processors and achieve optimal speed and efficiency. We also illustrate how to partition data and perform the algorithm on a LARPBS with fewer than  $mn^2$  processors, and hence show that the algorithm is highly scalable.

**Keywords:** LARPBS model, Hough transform, parallel algorithm

## 1. Introduction

The Hough transform is an important transformation in image processing and computer vision and has been widely used in the area of line detection, shape recognition and range alignment for moving image objects. Since the computational complexity of Hough transform is very large (it requires  $O(mn^2)$  time on a sequential machine), many approaches have been presented to speed up the computation, such as randomized Hough transform [6] (RHT). Many optimization algorithms such as genetic algorithm, simulated annealing and tabu search are used for Hough transform [15, 16]. Many parallel Hough transform algorithms on different architectures including mesh-tree [3], linear array [2], pyramid [4] and hypercube [12] interconnections are also presented to get high computation speed. Recently, several constant-time parallel Hough algorithms on reconfigurable mesh-connected machines are also proposed [1, 5, 8–11, 13]. Among them the ones described in [1, 11, 13] are the most efficient. For a problem with an  $n \times n$  images and  $m$   $\theta$ -values, the constant-time parallel algorithm on 3D reconfigurable mesh in [1] uses  $m \cdot n^3$  processors, the algorithm in [13] uses  $m^2 \cdot n^2$  processors, and the algorithm in [11] needs  $m \cdot (\log n)^2 \cdot n^2$

\*This research was supported in part by Chinese National Science Foundation under contract 60074013, Chinese National Foundation of High Performance Computing under contract 00219 and Science Foundation of Jiangsu Educational Commission, China, and in part by the U.S. National Science Foundation under Grants CCR-9503882, and ECS-0196569.

processors. Although these algorithms are faster than many other parallel Hough algorithms, they still require large number of processors and hence not efficient. A constant-time Hough transform algorithm on a 3D AROB (array with reconfigurable optical buses) using  $n^2m$  processors is presented in [14]. Since the 3D bus structure uses too many switches and optical buses, the algorithm requires much more hardware than a 1D optical bus structure our algorithm uses.

In this paper, we present a parallel algorithm for Hough transform a linear array with reconfigurable pipeline bus system (LARPBS). Suppose the number of  $\theta$ -values to be considered is  $m$ , for an image with  $n \times n$  pixels, the algorithm can complete Hough transform in  $O(1)$  time using  $mn^2$  processors and get the optimal speed and efficiency. Due to the advanced technology of VLSI and optical engineering, it is possible to integrate such large number of processors and electrooptic cells on a single chip. We also illustrate how to partition data and perform the algorithm on a LARPBS with fewer than  $mn^2$  processors, and hence show that the algorithm is also highly scalable.

## 2. Linear arrays with reconfigurable pipelined bus systems

Recently, arrays with reconfigurable optical bus systems [7] have been proposed and have drawn much attention from the researchers. In these systems, messages can be transmitted concurrently on a bus in a pipelined fashion and the bus can be reconfigured dynamically under program control to support different algorithmic requirements. LARPBS is one such model where any processor involvement is not allowed during a bus cycle, except setting switches up at the beginning of a bus cycle. Hence, it can exploit the high bandwidth of optical buses used to connect processors there. Many algorithms have been designed for basic data movement operations, sorting and selection, computational geometry, and PRAM simulation on the LARPBS model [7].

A pipelined optical bus system uses optical waveguides instead of electrical signals to transfer messages among processors. Besides the high propagation speed of light, optical signal transmission on an optical bus has other two important characteristics; they are unidirectional propagation and predictable propagation delay. These advantages of using waveguides enable synchronized concurrent accesses of an optical bus in a pipelined fashion. Such pipelined optical bus systems can support a massive volume of communications simultaneously and are particularly appropriate for applications that involve intensive communication operations such as broadcasting, one-to-one communication, multicasting, compression, split, and many irregular communication patterns.

It has been shown that by using the coincident pulse addressing technique, all the above primitive operations take  $O(1)$  bus cycles, where the bus cycle length is the end-to-end message transmission time over a bus [7]. To avoid controversy, let us emphasize that in this paper, by " $O(f(p))$  time" we mean  $O(f(p))$  bus cycles for communication plus  $O(f(p))$  time for local computation.

In addition to supporting fast communications, an optical bus itself can be used as a computing device for global aggregation. It was proven in [7] that by using  $n$  processors, the summation of  $n$  integers or reals with bounded magnitude and precision, the prefix sums

of  $n$  binary values, the logical-or and logical-and of  $n$  Boolean values can be calculated in constant number of bus cycles.

In addition to the tremendous communication capabilities, an LARPBS can also be partitioned into several independent subarrays. The subarrays can operate as regular linear arrays with pipelined optical bus systems, and all subarrays can be used independently for different computations without interference. Hence, this architecture is very suitable for many divide-and-conquer problems. The basic communication, data movement, and aggregation operations provide an algorithmic view on parallel computing using optical buses, and also allow us to develop, specify, and analyze parallel algorithms by ignoring optical and engineering details. These powerful primitives that support massive parallel communications plus the reconfigurability of optical buses make the LARPBS computing model very attractive in solving problems that are both computation and communication intensive. The reader is referred to [7] for more details on the LARPBS model and its basic operations.

### 3. The parallel Hough transform algorithm

For an  $n \times n$  binary image, and  $m$  values of  $\theta$ , we denote the pixels in the image as  $a(i, j)$ ,  $0 \leq i, j \leq n - 1$ . For line detection, a foreground pixel  $a(i, j)$  is transformed into a sinusoidal curve in the  $(\rho, \theta)$ -parameter plane which is defined as:

$$\rho = i \cos \theta + j \sin \theta$$

The pair  $(\rho, \theta)$  represents a parameterization of a line in the image space where  $\rho$  is the perpendicular distance from the origin to that line, and  $\theta$  is the slope of the perpendicular from the origin to the line. Each point  $(\rho, \theta)$  in  $(\rho, \theta)$ -plane corresponds to a line at angle  $\theta$  and distance  $\rho$  from the origin in the original data space. The value of a function in  $(\rho, \theta)$ -plane gives the point density along a line in the data space. For each point in the original space consider all the lines which go through that point at a particular discrete set of angles, chosen a priori. For each angle  $\theta$ , calculate the distance to the line through the point at that angle and discretise that distance using an a priori chosen discretisation, giving value  $\rho$ .

Make a corresponding discretisation of the  $(\rho, \theta)$ -plane—this will result in a set of boxes in  $(\rho, \theta)$ -plane. These boxes are used as accumulators. For each line we consider above, we increment a count (initialised at zero) in the  $(\rho, \theta)$ -plane at point  $(\rho, \theta)$ . After considering all the lines through all the points, an accumulator with a high value will probably correspond to a line of points. Namely, if the value of the  $(\rho, \theta)$  accumulator is larger than a given threshold, we acknowledge that there exists a line  $\rho = i \cos \theta + j \sin \theta$  in the image.

We partition the  $\theta$ -values in  $[0, \pi/4]$  into four parts which are in the domains of  $[0, \pi/4)$ ,  $[\pi/4, \pi/2)$ ,  $[\pi/2, 3\pi/4)$ , and  $[3\pi/4, \pi]$ , respectively. Without loss of generality, we only give the algorithm for the  $\theta$ -values in the part of  $[0, \pi/4)$  and the algorithm can be easily generalized to the other  $\theta$ -values. For the  $m/4$   $\theta$ -values in the part of  $[0, \pi/4)$ , we denote  $\theta_k = k/m \cdot \pi$ ,  $0 \leq k < m/4$ . Given a  $\theta$ -value and  $\rho$ -value, we use a counter  $\text{count}(\theta, \rho)$  to indicate the number of black pixels on the line  $(\theta, \rho)$ . For a  $\theta$ -value in  $[0, \pi/4)$ , let the

$\rho$ -value of the pixel  $a(i, j)$  be  $\rho_{a(i,j)}$ , from the geometric features and the relations of the pixels in an image, we have the following lemma:

**Lemma 1** For a  $\theta$ -value in  $[0, \pi/4)$  and integers  $i, j : 0 \leq i \leq n - 1, 0 \leq j \leq n - 3$ , if the  $\rho$ -values of the pixels  $a(i, j)$  and  $a(i, j + 2)$  satisfy:  $\rho_{a(i,j)} = \rho_{a(i,j+2)}$ , then we have:  $\rho_{a(i,j)} = \rho_{a(i,j+1)} = \rho_{a(i,j+2)}$

**Lemma 2** For a  $\theta$ -value in  $[0, \pi/4)$ , a fixed  $\rho$ -value and integers  $i, h, j : 0 \leq i, h \leq n - 1, 0 \leq j \leq n - 2$ , if both the pixels  $a(i, j)$  and  $a(h, j + 1)$  are on the line  $(\theta, \rho)$ , then there is no pixel in the image on line  $(\theta, \rho)$  between  $a(i, j)$  and  $a(h, j + 1)$ .

Lemma 1 illustrates that the pixels on the same row of an image with the same  $\rho$ -value are all consecutive pixels in the row. Lemma 2 tells us that for a  $\theta$ -value in  $[0, \pi/4)$  and a fixed  $\rho$ -value, we can detect all the pixels in line  $(\theta, \rho)$  thoroughly by scanning the image row by row.

Suppose there are  $N = m \cdot n^2$  processors in the array, and the indexes of the processors are  $0, 1, 2, \dots, N - 1$ , we denote the  $r$ -th processor in the array as  $p_r$ . A processor in the array also can be identified by a unique 3D index  $(k, i, j)$ . Let  $r = kn^2 + in + j$ ,  $p_r$  can also be denoted as  $p(k, i, j)$ . A subarray consisting of processors  $p(k, i, 0), p(k, i, 1), \dots, p(k, i, n - 1)$  is denoted as  $p(k, i, *)$ . Similarly, we use  $p(k, *, *)$  to denote the subarray consisting  $p(k, 0, *), p(k, 1, *), \dots, p(k, n - 1, *)$ .

In our algorithm, we first broadcast the pixel  $a(i, j)$  ( $0 \leq i, j \leq n - 1$ ) onto the processors in the subarray  $p(*, i, j)$ , and also broadcast the value of  $\theta_k$  onto the processors in  $p(k, *, *)$  ( $0 < k \leq m$ ) which is reconfigured as a independent subarray. In this way the whole array is reconfigured into  $m$  subarrays. Each subarray  $p(k, *, *)$  consists of  $n^2$  processors in which a copy of all the pixels of the image are stored. Processor  $p(k, i, j)$  in the subarray has the pixel  $a(i, j)$  and the value of  $\theta_k$ .

For a fixed  $\theta_k$ , there is a group of parallel lines whose slopes are  $h = \text{tg}(\theta_k + \pi/2) = -\text{ctg} \theta$ . Suppose the pixels on the  $x$ -axle are  $(0, 0), (1, 0), \dots, (M_1, 0), \dots, (n - 1, 0), (n, 0), \dots, (M_2, 0)$  as shown in Figure 1, where  $(M_1, 0)$  is the intersection of the line whose slope is  $h$

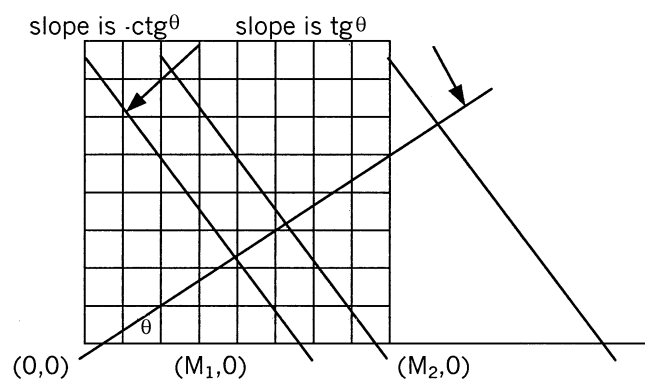


Figure 1. Lines and their head pixels.

passing the point  $(0, n-1)$  with the  $x$ -axle,  $(M_2, 0)$  is the intersection of the line whose slope is  $h$  passing the point  $(n-1, n-1)$  with the  $x$ -axle. It is obvious that  $M_1 = \lceil \frac{1}{h}(n-1) \rceil$ ,  $M_2 = \lceil (1 - \frac{1}{h})(n-1) \rceil$ . We call  $(0, 0), (1, 0), \dots, (M_1, 0), \dots, (n-1, 0), (n, 0), \dots, (M_2, 0)$  the head pixels of the lines. Let  $g_i$  be the number of all pixels in the image including both black and white ones on the line passing head pixel  $(i, 0)$  and with slope  $h$ . Suppose the highest and lowest  $y$  coordinates of all pixels on this line are  $u_i$  and  $l_i$ , respectively, then  $g_i = u_i - l_i + 1$ . To get the values of  $l_i$  and  $u_i$ , there are three different cases which should be considered:

- (1) If  $0 \leq i \leq M_1$ , then  $l_i = 0, u_i = \max \{k \in Z | (i + \frac{k}{h}) \geq 0\}$ ;
- (2) If  $M_1 < i \leq n-1$ , then  $l_i = 0, u_i = n-1$ ;
- (3) If  $n-1 < i \leq M_2$ , then  $l_i = \min \{k \in Z | (i + \frac{k}{h}) \leq n-1\}, u_i = n-1$ ;

Here,  $Z$  represents the set of integers. For every head pixel  $(i, 0)$ , we can get its  $l_i$  and  $u_i$  values by the formulas listed above, and then its  $g_i$  and  $\rho$ -value, which is denoted as  $\rho_i$ , can also be obtained. Then we can move all the pixels which belong to the same head pixel to a number of consecutive processors, and make the pixels belonging to the adjacent head pixels be located in successive processors. To implement these data movement, we first need to know the indexes of the destination processors which the head pixels will move to. We denote the index of the destination processor of the  $i$ -th head pixel as  $S_i$  ( $i = 0, 1, \dots, M_2$ ), which is also the index of the first processor in the group to which all the pixels belonging to the  $i$ -th head pixel will move. It is obvious that  $S_i = g_1 + g_2 + \dots + g_i$ , and we can get  $S_1, S_2, \dots, S_{M_2}$  by a prefix operation on  $g_1, g_2, \dots, g_{M_2}$ . To get the indexes of the destination processor of each pixel belonging to the head pixel  $a(i, 0)$ , we first broadcast parameters  $\rho_i, i, l_i, S_i$  to all the processors where these pixels are stored before moving, namely, to broadcast the parameters to the processors where pixels  $a(i + \lceil \frac{j}{h} \rceil, j)$  ( $l_i \leq j \leq u_i$ ) are stored before the moving operation. Suppose the parameters received by the processor where pixel  $a(x, y)$  is stored are  $\rho_i, i, l_i, S_i$ , then we know that pixel  $a(x, y)$  belongs to the line of the  $i$ -th head pixel, and it is the  $(y - l_i + 1)$ -th pixel in the line. Therefore the index of its destination processor is  $S_i + y - l_i + 1$ . All the pixels and their  $\rho$ -values can be transmitted to their destination processors so that all the pixels which belong to the same head pixel are sent to a number of consecutive processors, and the pixels belonging to the adjacent head pixels are located in successive processors.

After such data transmission, the  $n^2$  pixels are rearranged according to the order of their head pixels and their  $y$  coordinates. Although different head pixels may have identical  $\rho$ -value, since the head pixels are on the same row, by Lemma 1, we know they are consecutive head pixels. Therefore, all the pixels with same  $\rho$ -value are located in a number of consecutive processors. Each processor compares the  $\rho$ -value with the neighbor processor to its left. If the  $\rho$  of a processor is different from that of its left neighbor, we call it "boundary element". The processor array can be reconfigured into several subarrays by these boundary elements so that the pixels in the processors of one subarray are with identical  $\rho$ -value, and the pixels with the same  $\rho$ -value are all located in one subarray. Suppose the  $\rho$ -value of the pixels in the  $i$ -th subarray is  $\rho_i$ , the value of the counter count  $(\theta_k, \rho_i)$  of line  $(\theta_k, \rho_i)$  can be obtained by accumulating all the  $a(x, y)$  values in the subarray.

Memory locations  $A(k, i, j)$ ,  $\theta(k, i, j)$ ,  $\rho(k, i, j)$ ,  $l(k, i, j)$ ,  $S(k, i, j)$  are used in processor  $P(k, i, j)$  to store  $a(i, j)$ ,  $\theta_k$ , and its parameters  $\rho, l$  and  $S$ , respectively. Let  $r = kn^2 + in + j$ , they can also be denoted as  $A(r)$ ,  $\theta(r)$ ,  $\rho(r)$ ,  $l(r)$  and  $S(r)$ , respectively, when a 1D notation is used.

Our parallel Hough transform algorithm can be described as follows. The algorithm only considers the case of  $\theta \in [0, \pi/4]$ . The algorithm is very similar for the  $\theta$ -values in other domains.

**Algorithm** PARA-HOUGH( $n, m$ )

```

{ /* Initially,  $a(i, j)$  is stored in memory location  $A(0, i, j)$  of  $P(0, i, j)$  */
  for  $0 \leq k \leq m/4 - 1$  par-do
    { Part 1 for  $0 \leq i, j \leq n - 1$  par-do  $A(k, i, j) \leftarrow A(0, i, j)$ ;
      Part 2 /* The following steps are performed in subarray  $p(k, *, *)$  */
        Step 2.1 for  $0 \leq i, j \leq n - 1$  par-do  $\theta(k, i, j) = k \cdot \pi/m$ ;
        Step 2.2 for  $0 \leq i \leq M_2$  par-do
          /* Executed by processor  $p_{kn^2+i}$  which is the  $i$ th processor
            in subarray  $p(k, *, *)$  */
          {  $\rho_i = i \cdot \cos \theta_k$ ;
            if  $0 \leq i \leq M_1$  then {  $l_i = 0$ ;  $u_i = \max\{k \in Z | (i + \frac{k}{h}) \geq 0\}$  };
            if  $M_1 < i \leq n - 1$  then {  $l_i = 0$ ;  $u_i = n - 1$  };
            if  $n - 1 < i \leq M_2$  then {  $l_i = \min\{k \in Z | (i + \frac{k}{h}) \leq n - 1$ ;  $u_i = n - 1$  };
               $g_i = u_i - l_i + 1^\circ$ 
            }
          Step 2.3 /* Prefix computation in processors  $p_{kn^2+1}, p_{kn^2+2}, \dots, p_{kn^2+M_2}$  */
            for  $1 \leq i \leq M_2$  par-do  $S_i \leftarrow g_1 + g_2 + \dots + g_i$ 
          Step 2.4 /* Broadcast  $\rho_i, l_i, S_i$  to the processors where
            pixels  $a(i + \lceil \frac{i}{h} \rceil, j)$  are located */
            for  $1 \leq i \leq M_2$  par-do
              for  $l_i \leq j \leq u_i$  par-do
                {  $\rho(k, i + \lceil \frac{i}{k} \rceil, j) \leftarrow \rho_i$ ;  $l(k, i + \lceil \frac{i}{k} \rceil, j) \leftarrow l_i$ ;
                   $S(k, i + \lceil \frac{i}{k} \rceil, j) \leftarrow S_i$  }
          Step 2.5 for  $0 \leq i \leq n - 1$  par-do
            for  $0 \leq j \leq n - 1$  par-do
              {  $A(k, i, j) \rightarrow A[kn^2 + S(k, i + \lceil \frac{i}{k} \rceil, j) + j - l(k, i + \lceil \frac{i}{k} \rceil, j) + 1]$ ;
                 $\rho(k, i, j) \rightarrow \rho[kn^2 + S(k, i + \lceil \frac{i}{k} \rceil, j) + j - l(k, i + \lceil \frac{i}{k} \rceil, j) + 1]$ 
              }
          Step 2.6 for  $1 \leq i \leq n^2 - 1$  par-do
            {  $\rho(kn^2 + i) \rightarrow \sigma(kn^2 + i - 1)$ ;
              if  $\rho(kn^2 + i) = \sigma(kn^2 + i)$  then  $\tau(kn^2 + i) = 1$ 
                else  $\tau(kn^2 + i) = 0$ 
            }
          Step 2.7 Reconfigure the subarray  $p(k, *, *)$  into several smaller subarrays by
            processors whose  $\tau$  values are 0. In every new subarray,  $\rho$ -values in all
            the processors are identical. Suppose the  $\rho$ -value of the pixels in the  $i$ -th

```

subarray is  $\rho_i$ , accumulate all the  $a(x, y)$  values in the subarray to obtain  
the value of the counter count  $(\theta_k, \rho_i)$  of line  $(\theta_k, \rho_i)$ .

}

}

In the algorithm above, Part 1, Step 2.4 are operations of broadcast, Steps 2.1 and 2.2 are independent computations in each processors, Step 2.3 computes prefix sum of integers, Steps 2.5 and 2.6 are one-to-one communication, Step 2.7 is an operation of binary sum. Therefore the algorithm consists of 6 parallel primitive operations and a few parallel arithmetic operations. Each of these primitive operations can be accomplished in  $O(1)$  time. Therefore, for one  $\theta$ -value, the algorithm can be performed in constant time using  $n^2$  processors. For  $m/4$   $\theta$ -values in the domain of  $[0, \pi/4]$ , the algorithm can be performed in constant time using  $m \cdot n^2/4$  processors. Similar algorithm for  $\theta$ -values in other domains also can be performed in constant time. Therefore, using  $m \cdot n^2$  processors, the time complexity of our Hough transform algorithm is  $O(1)$  which has an optimal time. Obviously for an  $n \times n$  binary image and  $m$  values of  $\theta$ , the computational complexity can not be lower than  $O(mn^2)$ . Since all the communications are handled by the primitive operations each of which takes only  $O(1)$  time, the communication overhead is limited. Since each processor stores only one pixel, the algorithm has a good load balancing between processors and memory requirement reaches its minimum. Therefore our algorithm is also optimal in terms of efficiency and computational cost. To our best knowledge, this is the first constant-time, cost-optimal Hough transform algorithm on the LARPBS model.

#### 4. Scalability of the algorithm

While speed is an important motivation for parallel computing, there is another issue in realistic parallel computing, namely, scalability, which measures the ability to maintain speedup linearly proportional to the number of processors. For a large  $n \times n$  binary image, it is almost impractical to build an LARPBS system with  $m \cdot n^2$  processors. On a system with fixed size, a large image should be partitioned into smaller images to fit into the fix-sized array. We say that a parallel algorithm is scalable in the range  $[p_1, p_2]$ , if linear speedup can be achieved for all  $p_1 \leq p \leq p_2$ , where  $p$  is the number of processors used. In the other words, suppose the time complexity of a parallel algorithm be presented as  $O(T(n))$  using  $p$  processors where  $n$  is the size of the problem, for some constant  $r > 0$ , if  $p/r$  processors are used, the time complexity must be bounded by  $rO(T(n))$  in order to be scalable. Now we show that for  $1 \leq p \leq mn^2$ , our algorithm is scalable. When  $p < mn^2$ , the algorithm can also be executed using the method of partition, and the time complexity is  $O(mn^2/p)$ . Therefore, our algorithm is highly scalable.

(1) When  $n^2 \leq p \leq mn^2$ , let  $p = r \cdot n^2$ , then  $1 \leq r \leq m$ . Suppose  $q = \lceil m/r \rceil$ ,

We can partition the  $m$   $\theta$ -values into  $q$  groups each of which consists of  $r$   $\theta$ -values. Since there are  $p = r \cdot n^2$  processors in the array, one group of  $r$   $\theta$ -values can be processed in

$O(1)$  time. The array processes the  $q$  groups of  $\theta$ -values sequentially in  $O(q) = O(mn^2/p)$  time. Therefore, our algorithm is scalable when  $n^2 \leq p \leq mn^2$ .

- (2) When  $0 < p < n^2$ , let  $q = \lceil n^2/r \rceil$ . Since the number of processors is fewer than the number of pixels in the image, we can process the  $\theta$ -values sequentially.

In the process of one  $\theta$ -value, Part 1 of the algorithm, which distributes the pixels to the array processors, should be modified as follows: Partition the  $n^2$  pixels in the image into  $q$  groups each of which consists of  $p$  pixels. These pixels are distributed into the array in the following way: pixel  $a(i, j)$  is stored as the  $\lceil \frac{(i-1)n+j}{p} \rceil$ -th pixel in the  $[((i-1)n+j) \bmod p]$ -th processor.

Each step in Part 2 of the algorithm can be partitioned and executed as follows.

Step 2.1 calculates the  $\theta$ -value. Since all the processors have an identical  $\theta$ -value, the computation can be carried out independently in each processor, and the  $\theta$ -value obtained can be shared by the pixels stored in the same processor.

Step 2.2 calculates the parameters  $\rho_i, l_i, u_i$ , and  $g_i$  for head pixels. If  $p \geq M_2$ , then the first  $M_2$  processors in the array are used to compute these parameters. If  $p < M_2$ , then  $M_2$  head pixels are divided into  $p$  groups each of which has  $\lceil M_2/p \rceil$  head pixels. Since the parameter calculations for the different head pixels are independent, parameter calculations for  $\lceil M_2/p \rceil$  head pixels in each group can be performed in one processor sequentially.

Step 2.3 calculates the prefix sum of  $g_1, g_2, \dots$ , and  $g_{M_2}$ . If  $p \geq M_2$ , then this operation of prefix sum can be accomplished using the first  $M_2$  processors in the array. If  $p < M_2$ , then  $M_2$   $g$ -values are divided into  $\lceil M_2/p \rceil$  groups each of which consists of  $pg$ -values which are distributed to the  $p$  processors of the array in the following fashion. Parameter  $g_i$  is assigned as the  $\lceil \frac{i}{p} \rceil$ -th  $g$ -value of the  $[(i-1) \bmod p]$ -th processor. Then the array calculates the prefix sums of the  $g$ -values in the groups sequentially. First the prefix sums  $S_i = \sum_{j=1}^i g_j$  ( $i = 1, 2, \dots, p$ ) of the  $g$ -values in the first group is calculated. Then the array calculates the local prefix sums  $sum_i = \sum_{j=p+1}^i g_j$  ( $i = p+1, \dots, 2p$ ) of the  $g$ -values in the second group. The global prefix sums of the  $g$ -values in the second group can be obtained using the local prefix sums:  $S_i = \sum_{j=1}^i g_j = \sum_{j=1}^p g_j + \sum_{j=p+1}^i g_j = S_p + sum_i$ . The prefix sums of the  $g$ -values in all the other groups can be computed similarly.

Step 2.4 broadcasts parameters  $\rho_i, l_i, u_i$ , and  $S_i$  to the processors where pixels  $a(i + \lceil \frac{i}{h} \rceil, j)$  are stored. Noticing that pixel  $a(i, j)$  is the  $\lceil \frac{(i-1)n+j}{p} \rceil$ -th pixel stored in the  $[(i-1)n+j) \bmod p]$ -th processor, these parameters can be broadcasted to their destination processors. Since each processor has  $n^2/p$  pixels, each processor will receive at most  $n^2/p$  sets of parameters. It takes at most  $O(n^2/p)$  time for a processor to receive these parameters.

Step 2.5 rearranges the pixels in the processors according to the order of their head pixels and their  $y$  coordinates. Since the number of the processors  $p$  is fewer than the number of pixels  $n^2$ , we arrange the pixels in groups as follows. For pixel  $A(k, i, j)$ , suppose it is the  $d$ -th pixel after rearranging, then

$$d = kn^2 + S\left(k, i + \left\lceil \frac{j}{h} \right\rceil, j\right) + j - l\left(k, i + \left\lceil \frac{j}{h} \right\rceil, j\right) + 1.$$

It means pixel  $A(k, i, j)$  should be sent to the  $(d \bmod p)$ -th processor as its  $(\lceil \frac{d}{p} \rceil)$ -th pixel. The pixels are transmitted in the order of their groups. Since there are  $n^2/p$  groups, it will take  $O(n^2/p)$  time to accomplish the data transmission.

Steps 2.6 and 2.7 counts the number of the black pixels with the same  $\rho$ -value. Since the number of processors  $p$  is fewer than the number of pixels  $n^2$ , we can count the pixels in the order of the groups. First, we count the pixels in the first group which consists of all the first pixels of the processors. Suppose the last  $\rho$ -value counted in the first group is  $\rho_{\text{last1}}$ , and the number of black pixels in this group on line  $(\theta_k, \rho_{\text{last1}})$  is  $\text{count}_1(\theta_k, \rho_{\text{last1}})$ . If there are some pixels in the second group of pixels with  $\rho_{\text{last1}}$  as their  $\rho$ -value, after the second group of pixels are counted, suppose the number of these pixels is  $\text{count}_2(\theta_k, \rho_{\text{last1}})$ , this number should be added to  $\text{count}_1(\theta_k, \rho_{\text{last1}})$ :

$$\text{count}_1(\theta_k, \rho_{\text{last1}}) := \text{count}_1(\theta_k, \rho_{\text{last1}}) + \text{count}_2(\theta_k, \rho_{\text{last1}})$$

Similarly, in the process of the other groups, the count number of the last  $\rho$ -value of each group must be modified by the count numbers of the same  $\rho$ -value in the latter groups.

Now we analyze the time complexity of Part 2 executed using the method of partition illustrated above. Step 2.1 needs  $O(1)$  time since the computation of the  $\theta$ -value can be carried out independently in each processor. In Step 2.2, if  $p \geq M_2$ , the parameters can be calculated in the first  $M_2$  processors of the array in  $O(1)$  time. If  $p < M_2$ , every processor calculates the parameters for  $M_2/p$  head pixels in  $O(M_2/p) = O(n/p)$  time. In Step 2.3, if  $p > M_2$ , the prefix operation can be performed by the first  $M_2$  processors in the array in  $O(1)$  time. If  $p \leq M_2$ , every processor should carry out  $M_2/p$  prefix operations which needs  $O(M_2/p) = O(n/p)$  time. Step 2.4 broadcasts the parameters to the processors where the pixels are stored. Since there are at most  $n^2/p$  pixels stored in each processor, time complexity for this step is  $O(n^2/p)$ . Step 2.5 rearranges the pixels. Since each processor receives at most  $n^2/p$  pixels, time complexity for this step is also  $O(n^2/p)$ . Steps 2.6 and 2.7 count the number of the black pixels with the same  $\rho$ -value in the order of the groups of pixels. Since there are  $n^2/p$  groups, the time required for this step is  $O(n^2/p)$ . Therefore for one  $\theta$ -value,  $O(n^2/p)$  time is needed in the array with  $p$  processors. For  $m$   $\theta$ -values, time complexity is  $O(mn^2/p)$ . In the execution of the algorithm no extra communications and arithmetic operations are involved. In addition, in the execution of the prefix sum operation in Step 2.3 and the summation computation in Step 2.6, since the prefix or summation operations of  $n^2/p$  values are performed in one processor, the amount of communications is greatly reduced.

From the analysis of parts (1) and (2) above, we can see that for an  $n \times n$  image and  $m$   $\theta$ -values, when  $1 \leq p \leq mn^2$ , our algorithm can be executed using the method of partition illustrated above using  $p$  processors in  $O(mn^2/p)$  time. Therefore our algorithm is highly scalable. The cost of the algorithm execution is  $O(p \cdot mn^2/p) = O(mn^2)$ , which also reaches the optimal in terms of cost and efficiency. Compared with the Hough algorithm on a 3D AROB by Pavel et al. [14] which also uses  $n^2m$  processors, since the 3D bus structure uses too many switches and optical buses, the algorithm requires much more hardware than a 1D optical bus structure which our algorithm uses. Furthermore, in their algorithm, since one copy of the pixels in the image are stored in each 2D subarray, the switches are set

according to the  $\theta$ -values assigned to the subarray so that the processors with pixels located at the same line with the  $\theta$ -value are connected. This makes it difficult to partition the pixels stored in each 2D subarray. Therefore, the algorithm is scalable only when the number of processor  $p \geq n^2$ , while our algorithm is completely scalable when  $1 \leq p \leq mn^2$ . This makes their algorithm still need a large number of processors. In addition, our algorithm on the 1D LARPBS model is much more sophisticated than their algorithm on the 3D AROB model since we have to avoid a lot of conflicts in communication on the LARPBS model to achieve optimal algorithm while it is much easier to do so on the 3D AROB model.

## 5. Conclusion

A parallel algorithm for Hough transform on linear array with reconfigurable pipeline bus system (LARPBS) is presented. Suppose the number of  $\theta$ -values to be considered is  $m$ , for an image with  $n \times n$  pixels, the algorithm exploits many geometric features of the problem and can complete Hough transform in  $O(1)$  time using  $mn^2$  processors and get the optimal speed and efficiency. To our knowledge, this is the fastest and most efficient parallel Hough transform algorithm reported so far on the LARPBS model. We also illustrate how to partition and perform the algorithm on a LARPBS with fewer than  $mn^2$  processors, and hence show the algorithm is highly scalable. Our algorithm takes advantage of many important merits of LARPBS such as its high communication bandwidth, the versatile communication patterns it supports, and its ability of utilizing communication reconfigurability as an integral part of a parallel computation. It is shown that the LARPBS is a powerful architecture for exploiting large degree of parallelism in a computational problem that most other machine models cannot achieve.

## References

1. K. L. Chung and H. Y. Lin. Hough transform on reconfigurable meshes. *Computer Vision and Image Understanding*, 61(2):278–284, 1995.
2. A. L. Fisher and P. T. Highnam. Computing the Hough transform on a scan line array processor. *IEEE Transactions on PAMI*, 11(3):262–265, 1989.
3. H. A. H. Ibrahim, J. R. Kender, and D. E. Shaw. The analysis and performance of two middle-level vision tasks on a fine-grained SIMD tree machine. *Proceedings IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1985, pp. 387–393.
4. J. Jolion and A. Rosenfeld. An  $O(\log n)$  pyramid Hough transform. *Pattern Recognition Letters*, 9:343–349, 1989.
5. T. W. Kao, S. J. Horng, and Y. L. Wang. An  $O(1)$  time algorithm for computing histogram and the Hough transform on a cross-bridge reconfigurable array of processors. *IEEE Transactions on Systems, Man and Cybernetics*, 25(4):681–687, 1995.
6. Xu Lei and Oja Erkki. Randomized Hough transform (RHT): Basic mechanism, algorithms, and computational complexities. *CVGIP: Image Understanding*, 57(2):131–154, 1993.
7. K. Li, Y. Pan, and S.-Q. Zheng, eds. *Parallel Computing Using Optical Interconnections*, Kluwer Academic Publishers, Boston, USA, Hardbound, Oct. 1998, ISBN 0-7923-8296-X.
8. S. S. Lin. Constant-time Hough transform on the processor array with reconfigurable bus systems. *Computing*, 52:1–15, 1994.
9. M. Merry and J. W. Baker. Constant time algorithm for computing Hough transform on a reconfigurable mesh. *Image and Vision Computing*, 14:35–37, 1996.

10. Y. Pan. A more efficient constant time algorithm for computing the Hough transform. *Parallel Processing Letters*, 4(1/2):45–52, 1994.
11. Y. Pan. Constant-time Hough transform on a 3D reconfigurable mesh using fewer processors. *Proceedings of Reconfigurable Architectures Workshop (RAW2000)*, May 5, 2000, vol. 1800, LNCS, pp. 966–973.
12. Y. Pan and H. Y. H. Chung. Faster line detection algorithms on enhanced mesh connected arrays. *IEEE Proceeding-E*, 140(2):95–100, 1993.
13. Y. Pan, K. Li, and M. Hamdi. An improved constant time algorithm for computing the Radon and Hough transforms on a reconfigurable mesh. *IEEE Transactions on Systems, Man, and Cybernetics (Part A)*, 29(4):417–421, 1999.
14. S. Pavel and S. G. Akl. Computing the Hough transformation on arrays with reconfigurable optical buses. In K. Li, Y. Pan, and S. Q. Zheng, eds., *Parallel Computing Using Optical Interconnections*, Kluwer Academic Publishers, Boston, USA, Hardbound, Oct. 1998, pp. 205–226, ISBN 0-7923-8296-X.
15. G. Roth and M.D. Levin. Geometric primitive extraction using a genetic algorithm. *IEEE Trans on PAMI*, 9(16):901–905, 1994.
16. Jiang Tianzi and Ma Songde. Geometric primitive extraction using Tabu search. *Proc ICPR'96*, Vienna, Austria, 1996, vol. 2, pp. 266–279.