1989

Computing Hough Transforms on Hypercube Multicomputers

Sanjay Ranka
Syracuse University

Sartaj Sahni

SU-CIS-89-06

Computing Hough Transforms on Hypercube Multicomputers

Sanjay Ranka Sartaj Sahni

1989

School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

Abstract

Efficient algorithms to compute the Hough transform on MIMD and SIMD hypercube multicomputers are developed. Our algorithms can compute p angles of the Hough transform of an $N \times N$ image, $p \leq N$, in $0(p + \log N)$ time on both MIMD and SIMD hypercubes. These algorithms require $0(N^2)$ processors. We also consider the computation of the Hough transform on MIMD hypercubes with a fixed number of processors. Experimental results on an NCUBE/7 hypercube are presented.

Keywords and Phrases

Hough transform, MIMD and SIMD hypercube multicomputers, complexity

1 Introduction

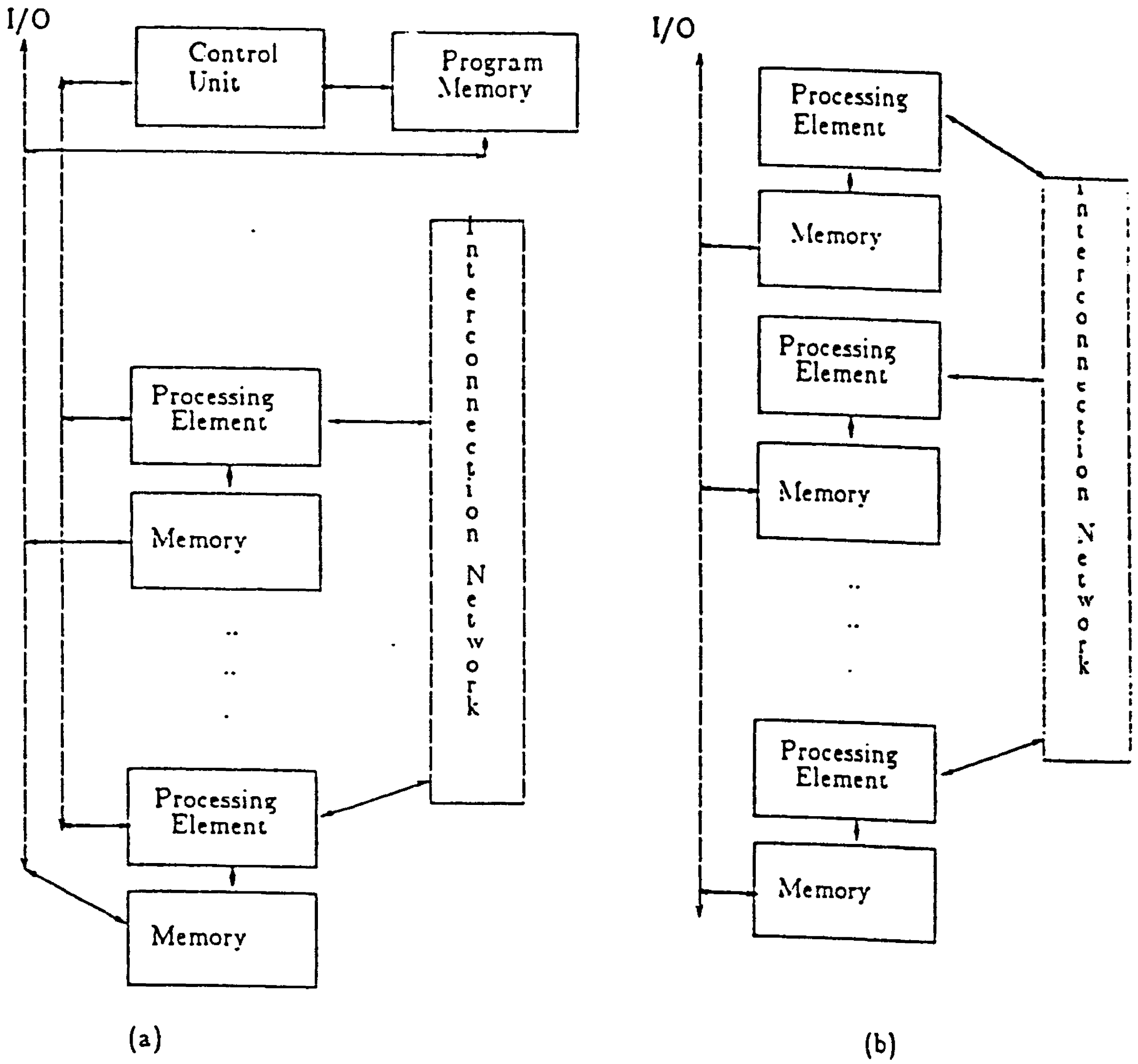
Hough transforms are used to detect straight lines or edges in an image. Let $I[0 \ldots N-1, 0 \ldots N-1]$ be an $N \times N$ image such that $I[x,y] = 1$ iff the image point $[x,y]$ is a possible edge point. $I[x,y] = 0$ otherwise. The p angle Hough transform of I is the array H such that

$$H[i,j] = |\{(x,y)|i = \lfloor x\cos\theta_j + y\sin\theta_j\rfloor, \theta_j = \frac{\pi}{p}(j+1) \text{ and } I[x,y] = 1\}|.$$

j takes on the integer values $0, 1, \ldots, p-1$. These correspond to the p angles $\theta_j = \frac{\pi}{p}(j+1), 0 \leq j < p$. Hence $0 < \theta_j \leq \pi$. For θ_j in this range and x and y in the range $0 \ldots N-1$, $\lfloor x\cos\theta_j + y\sin\theta_j\rfloor$ is in the range $-\sqrt{2}N \ldots \sqrt{2}N$. Hence H is at most a $2\sqrt{2}N \times p$ matrix.

The serial algorithm to compute H has complexity $0(N^2p)$. Parallel algorithms to compute H have been developed by several researchers. Rosenfeld, Ornelas, and Hung [ROSE88], Cypher, Sanz, and Snyder [CYPH87], Guerra and Hambrush [GUERR87], and Sildberg [SILD86] consider mesh connected multicomputers; Fishburn and Highnam [FISH87] consider scan line array processors; Ibrahim, Kender, and Shaw [IBRA86] consider SIMD tree machines; and Chandran and Davis [CHAN87] consider the use of the Butterfly and Ncube multicomputers to compute the Hough transform.

In this paper we develop algorithms to compute the Hough transform on hypercube multicomputers. First, in Section 2, we describe our model for fine-grained MIMD and SIMD hypercubes and how to perform certain fundamental data movement operations on a hypercube. These are used in our subsequent development of hypercube algorithms for the Hough transform. In Section 3 we describe our Hough transform algorithm for an MIMD hypercube. The case of an SIMD hypercube is considered in Section 4. Section 5 considers the computation of the Hough transform on a medium-grained MIMD hypercube. Experimental results on an NCUBE/7 hypercube are also presented in this section.



(a) SIMD Hypercube (b) MIMD Hypercube

Figure 1: Hypercube Multicomputers

2 Preliminaries

2.1 Hypercube Multicomputer

Block diagrams of an SIMD and MIMD hypercube multicomputer are given in Figures 1(a) and 1(b), respectively. The important features of an SIMD hypercube and the programming notation we use are:

1. There are $p = 2^p$ processing elements connected together via a hypercube interconnection network (to be described later). Each PE has a unique index in the range $[0, 2^p - 1]$. We shall use brackets ([ ]) to index an array and parentheses (()) to index PEs. Thus $A[i]$ refers to the i'th element of array A and A(i) refers to the A register of PE i. Also, A[j](i) refers to the j'th element of array A in PE i. The local memory in each PE holds data only (i.e., no executable instructions). Hence PEs need to be able to perform only the basic arithmetic operations (i.e., no instruction fetch or decode is needed).

2. There is a separate program memory and control unit. The control unit performs instruction sequencing, fetching, and decoding. In addition, instructions and masks are broadcast by the control unit to the PEs for execution. An *instruction mask* is a boolean function used to select certain PEs to execute an instruction. For example, in the instruction

   $$A(i) := A(i) + 1, \qquad (i_0 = 1)$$

   $(i_0 = 1)$ is a mask that selects only those PEs whose index has bit 0 equal to 1; i.e., odd indexed PEs increment their A registers by 1. Sometimes we shall omit the PE indexing of registers. The above statement is therefore equivalent to the statement:

   $$A := A + 1, \qquad (i_0 = 1).$$

3. The topology of a 16-node hypercube interconnection network is shown in Figure 2. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}\ldots i_0$ be

the binary representation of the PE index i. Let $\bar{i}_k$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit; i.e., processor $i_{p-1}i_{p-2}\ldots i_0$ is connected to processors $i_{p-1}\ldots\bar{i}_k\ldots i_0, 0 \leq k \leq p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

4. Interprocessor assignments are denoted using the symbol ←, while intraprocessor assignments are denoted using the symbol :=. Thus the assignment statement:

$$B(i^{(2)}) \leftarrow B(i), \quad (i_2 = 0)$$

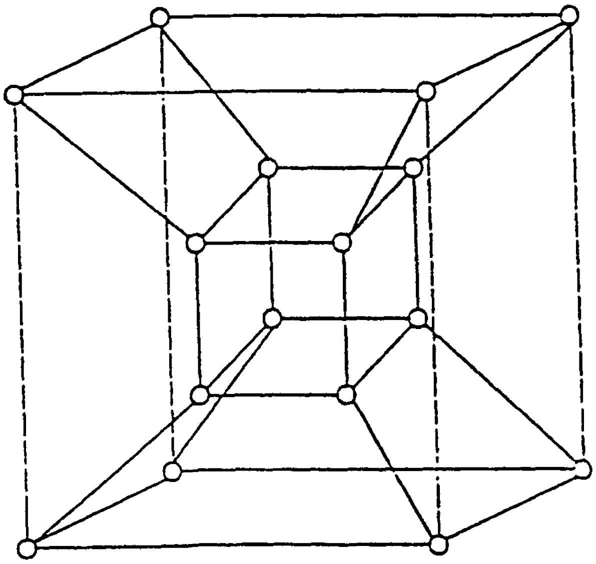


Figure 2: A 16-Node Hypercube (Dimension =4)

is executed only by the processors with bit 2 equal to 0. These processors transmit their B register data to the corresponding processors with bit 2 equal to 1.

5. In a *unit route*, data may be transmitted from one processor to another if it is directly connected. We assume that the links in the interconnection network are unidirectional. Hence, at any given time, data can be transferred either from PE $i(i_b = 0)$ to PE $i^{(b)}$ or from PE $i(i_b = 1)$ to PE $i^{(b)}$. Thus the instruction

$$B(i^{(2)}) \leftarrow B(i), (i_2 = 0)$$

takes one unit route, while the instruction

$$B(i^{(2)}) \leftarrow B(i)$$

takes two unit routes.

6. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

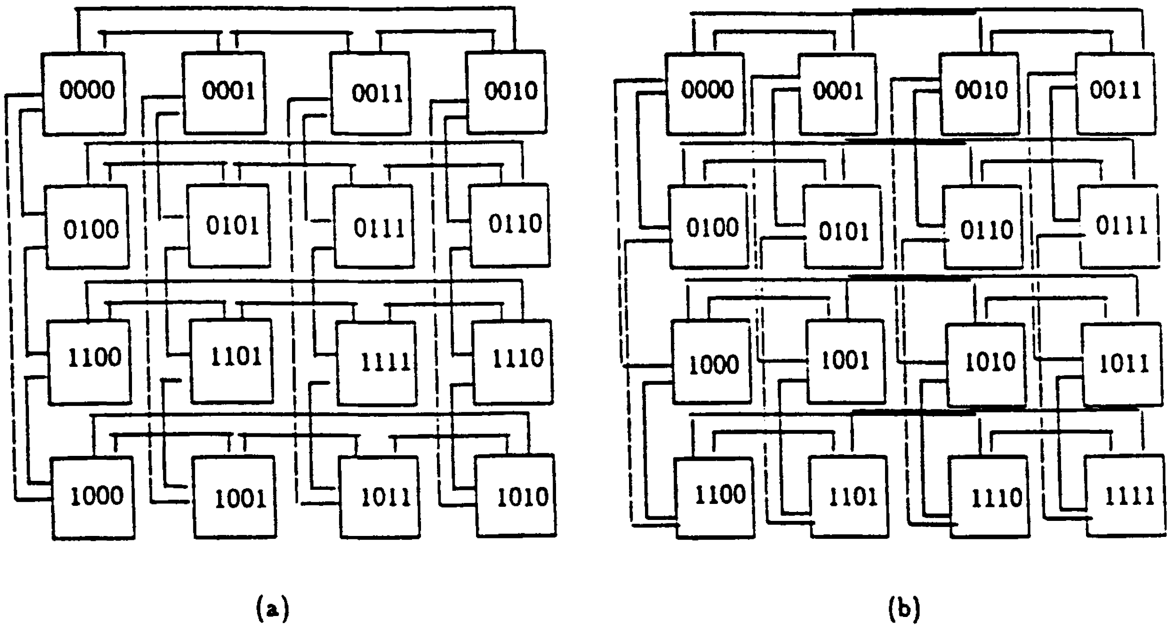
The features, notation, and assumptions for MIMD hypercubes differ from those of SIMD hypercubes in the following way:

There is no separate control unit and program memory. The local memory of each PE holds both the data and the program that the PE is to execute. At any given instance, different PEs may execute different instructions. In particular, PE i may transfer data to PE $i^{(b)}$, while PE j simultaneously transfers data to PE $j^{(a)}, a \neq b$.

2.2 Image Mapping

Figure 3(a) gives a two-dimensional grid interpretation of a dimension 4 hypercube. This is the binary-reflected gray code mapping of [CHAN86]. An i bit binary gray code S^i is defined recursively as below:

$$S_1 = 0, 1; \; S_k = 0[S_{k-1}], 1[S_{k-1}]^R$$



(a) gray code mapping

(b) row major mapping

Figure 3: Mapping of the Image

where $[S_{k-1}]^R$ is the reverse of the $k-1$ bit code S_{k-1} and $b[S]$ is obtained from S by prefixing b to each entry of S. So, S_2 = 00, 01, 11, 10 and S_3 = 000, 001, 011, 010, 110, 111, 101, 100.

If $N = 2^n$, then S_{2n} is used. The elements of S_{2n} are assigned to the elements of the $N \times N$ grid in a snakelike row major order [THOM77]. This mapping has the property that grid elements that are neighbors are assigned to neighboring hypercube nodes.

Figure 3(b) shows an alternate embedding of a 4×4 image grid into a dimension 4 hypercube. The index of the PE at position (i, j) of the grid is obtained using the standard row major mapping of a two-dimensional array onto a one-dimensional array [HORO85]. I.e., for an $N \times N$ grid, the PE at postion (i, j) has index $iN + j$. Using the mapping, a two-dimensional image grid $I[0 \dots N, 0 \dots N]$ is easily mapped onto an N^2 hypercube (provided N is a power of 2) with one element of I per PE. Notice that, in this mapping, image elements that are neighbors in I (i.e., to the north, south, east, or west of one another) may not be neighbors (i.e., may not be directly connected) in the hypercube. This does not lead to any difficulties in the algorithms we develop.

We will assume that images are mapped using the gray code mapping for all MIMD algorithms and the row major mapping for all SIMD algorithms.

2.3 Basic Data Manipulation Operations

2.3.1 SIMD *SHIFT*

$SHIFT(A, i, W)$ shifts the A register data circularly counter-clockwise by i in windows of size W; i.e., $A(qW+j)$ is replaced by $A(qW+(j-i) \bmod W), 0 \leq q < (p/W)$. $SHIFT(A, i, W)$ on an SIMD computer can be performed in $2 \log W$ unit routes [PRAS87]. A minor modification of the algorithm given in [PRAS87] performs $i = 2^m$ shifts in $2 \log(W/i)$ unit routes [RANK88]. The wraparound feature of this shift operation is easily replaced by an end-off *zero* fill feature. In this case, $A(qw + j)$ is replaced by $A(qW + j - i)$ as long as $0 \leq j - i < W$, and by 0 otherwise. This change does not increase the number of unit routes. The end-off shift will be denoted $ESHIFT(A, i, W)$.

2.3.2 MIMD *SHIFT*

When i is a power of 2, $SHIFT(A, i, W)$ on an MIMD computer can be performed in $O(1)$ unit routes. An MIMD shift of 1 takes 1 unit route, of 2 takes 2 unit routes, of $N/2$ takes 4, and the remaining power of 2 shifts take 3 routes each. For any arbitrary i the shift can be completed in $3(\log W)/2 + 1$ unit routes on an MIMD computer [RANK88]. As in the case of the SIMD shift, the MIMD shift is also easily modified to an end-off *zero* fill shift without increasing the number of unit routes.

2.3.3 Data Circulation on an SIMD Hypercube

The data in the A registers of each of the R processors in an R processor subhypercube is to be circulated through each of the remaining R-1 PEs in the subhypercube. This can be accomplished using R-1 unit routes. The circulation algorithm uses the exchange sequence $X_r, R = 2^r$ defined recursively as [DEKE81]:

$$X_1 = 0,\ X_q = X_{q-1}, q-1, X_{q-1}(q > 1)$$

This sequence essentially treats a q-dimensional hypercube as two $q-1$-dimensional hypercubes. Data circulation is done in each of these in parallel using X_{q-1}. Next, an exchange is done along bit $q-1$. This causes the data in the two halves to be swapped. The swapped data is again circulated in the two half hypercubes using X_{q-1}. Let $f(r, i)$ be the i^{th} number (left to right) in the sequence $X_r, 1 \leq i < 2^r$. The resulting SIMD data circulation algorithm is given in Figure 4. Here, it is assumed that the r bits that define the subhypercube are bits $0, 1, 2, \ldots, r-1$. Because of our assumption of unidirectional links, each iteration of the *for* loop of Figure 4 takes 2 unit routes. Hence Figure 4 takes $2(R-1)$ unit routes. The function f can be computed by the control processor in $0(R)$ time and saved in an array of size $R-1$ (actually it is convenient to compute f on the fly using a stack of

```
procedure CIRCULATE(A);
    [data circulation]
 for $i := 1$ to $R - 1$ do
        $A(j^{f(r,i)}) \leftarrow A(j)$;
end
```

Figure 4: Data circulation in an SIMD hypercube

height log R). The following Lemma allows each processor to compute the origin of the current A value.

Lemma 1: [RANK88] Let $A_0, A_1, \ldots, A_{2^r-1}$ be the values in $A(0), A(1), \ldots, A(2^r - 1)$ initially. Let *index*(j, i) be such that A[*index*(j, i)] is in $A(j)$ following the i'th iteration of the *for* loop of Figure 4. Initially, *index*$(j, 0) = j$. For every $i, i > 0$,*index*$(j, i) =$*index*$(j, i - 1)\theta 2^{f(r,i)}$ (θ is the exclusive or operator).

2.3.4 Data Accumulation on MIMD Hypercube

For this operation, PE j has an array $A[0 \ldots M - 1]$ of size M. In addition, each PE has a value in its I register. After the data accumulation, the M elements of A in each PE j are such that:

$$A[i] \text{ (gray } (j)) = I \text{ (gray } ((j + i) \bmod P)), 0 \leq i < M, 0 \leq j < P.$$

This can be accomplished in $M - 1$ unit routes (for $P > 2$) by repeatedly shifting by -1 in windows of size P. The algorithm is given in Figure 5.

```
procedure ACCUM(A,I,M)
{each PE accumulates in $A$, the $I$ values of the next
$M$ PEs, including itself; P is the window size}
begin
  $A[0] := I$;
  for $i := 1$ to $M - 1$ do
  begin
    $SHIFT(I, -1, P)$ :
    $A[i] := I$;
  end
end {ACCUM}
```

Figure 5: Data accumulation

2.3.5 Data Accumulation on SIMD Hypercube

After the data accumulation, the M elements of A in each PE j are such that:

$$A[i](j) = I((j+i) \bmod P),\ 0 \leq i < M,\ 0 \leq j < P.$$

Data accumulation may be done efficiently by modifying the data circulation algorithm. It can be completed in $2(M-1) + \log_2(N/M)$ unit routes on an SIMD hypercube.

2.4 Initial and Final Configurations

We shall explicitly consider the computation of $H(i,j)$ only for $i > 0$. The computation for the case $i \leq 0$ is similar. Hence i is in the range $[0, \sqrt{2}N)$ and j is in the range $[0, p)$. We assume that N is a power of two and that $2N^2$ PEs are available. These are viewed as an $N \times 2N$ array as discussed in §2.2 for SIMD and MIMD hypercubes. Actually, only $N \times \sqrt{2}N$ PEs are needed; however, a hypercube must have a power of 2 processors. Furthermore, it is assumed that p divides N.

The image pixel $I[i,j]$ is initially stored in PE $[i,j]$ $0 \leq i,j < N$ in the above array view. $H[i,j]$ is stored in PE$[j,i]$ on completion.

3 MIMD Algorithm

Conceptually, our algorithm is similar to that of Cypher and Sanz [CYPHS8]. It computes the Hough transform in $0(p+N)$ time on an $N \times N$ SIMD mesh connected computer. We show how this algorithm can be mapped onto an MIMD hypercube with $2N^2$ processors. The complexity of the resulting hypercube algorithm is $O(p+\log N)$.

For simplicity, we divide the computation of H$[i,j]$, $i > 0$, $0 \leq j < p$ into four parts. These, respectively, correspond to the cases $0 \leq j < p/4$, $p/4 \leq j < p/2$, $p/2 \leq j < 3p/4$, and $3p/4 \leq j < p$. First, consider the case $p/4 \leq j < p/2$. Now, $\pi/4 < \theta_j \leq \pi/2$. The following two lemmas will suggest a computational scheme for this case.

Lemma 3.1: When $\pi/4 < \theta_j \leq \pi/2$, two pixels (x,y) and $(x,y+z), z > 0$, can contribute to the count of the same H$[i,j]$ only if $z = 1$.

Proof: If (x,y) and $(x,y+z)$ both contribute to the count of H$[i,j]$, then

$$i = \lfloor x\cos\theta_j + y\sin\theta_j\rfloor = \lfloor x\cos\theta_j + (y+z)\sin\theta_j)\rfloor$$

for some j, $p/4 \leq j < p/2$. Hence

$$(y+z)\sin\theta_j - y\sin\theta_j \leq 1$$

$$\text{or } z\sin\theta_j \leq 1.$$

Since $\pi/4 < \theta_j \leq \pi/2$, $\sin\theta_j > \sin\pi/4 > 0{\cdot}5{\cdot}$ Since z is a positive integer, only $z = 1$ can satisfy the relation $z\sin\theta_j \leq 1$.

Lemma 3.2: When $\pi/4 < \theta_j \leq \pi/2$, two pixels (x, y) and $(x+1, z)$ can contribute to the count of the same $H[i,j]$ only if $z \in \{y, y-1\}$.

Proof: If (x, y) and $(x+1, z)$ contribute to the same $H[i,j]$, then $i = \lfloor x\cos\theta_j + y\sin\theta_j \rfloor = \lfloor (x+1)\cos\theta_j + z\sin\theta_j \rfloor$.

So, $|(x+1)\cos\theta_j - x\cos\theta_j + (z-y)\sin\theta_j| \leq 1$

$$\begin{aligned}
&\text{or } |\cos\theta_j + (z-y)\sin\theta_j| \leq 1\\
&\text{or } |\cot\theta_j + (z-y)| \leq \text{cosec}\theta_j\\
&\text{or } -\text{cosec}\theta_j - \cot\theta_j \leq z - y \leq \text{cosec}\theta_j - \cot\theta_j
\end{aligned}$$

Since y and z are integers and θ_j is in the above range, it follows that $-1 \leq z-y \leq 0$. Hence $z \in \{y, y-1\}$. □

The computation of $H[i,j]$ for $i > 0$ and $\pi/4 \leq \theta_j < \pi/2$ can be done in two phases. In the first, subhypercubes of size $p \times 2N$ compute

$h[i,j] = |\{(x,y)|i = \lfloor x\cos\theta_j + y\sin\theta_j \rfloor,\ \pi/4 \leq \theta_j < \pi/2,\ I[x,y] = 1$, and (x, y) is in this subhypercube.

In the second phase, the $h[i,j]$ values from the different subhypercubes are summed to get

$$H[i,j] = \sum_{subhypercubes} h[i,j],\ i > 0,\ p/4 \leq j < p/2.$$

The phase 1 algorithm for each PE in a $p \times 2N$ subhypercube is given in Figure 6. In this algorithm, $[x, y]$ denotes a PE index relative to the whole $N \times 2N$ hypercube and $[w, y]$ denotes the index of the same PE relative to the $p \times 2N$ subhypercube it is in. Note that $w = x \bmod p$.

```
for $\ell := 0$ to $5p/4 - 1$ do
    if $(w = 0)$ and $(\ell < p/4)$ then
    [{row 0 initiates next $\theta_j$}
      create a record $Z = (i, j,$ sine, cosine, $q)$
      with
      sine=sin($\theta$), cosine= cos($\theta$), where $\theta = \frac{\pi}{p}(p/2 - \ell + m)$
      $i = \lfloor x$ cosine $+ y$ sine$\rfloor$, $j = p/2 - \ell - 1$
      $q = I[x, y]]$
    else [if max$\{1, \ell - p/4 + 1\} \leq w \leq \ell$ and $y < N$
      then {add in this PE's contribution}
        [Let $Z$ be the record received from PE$[w - 1, y]$
        Let $i' = \lfloor x$ cosine $+y$ sine $\rfloor$ and $q' = I[x, y]$
        if $i = i'$ then set $q = q + q'$
        else if $i = \phi$ then set $i = i'$ and $q = q'$
          else  [send $q$ to PE$[x, (y - 1)$ mod $2N]$
            set $Z = (i', j,$ sine, cosine $, q')]$
        if a $q$ is received from PE$[x, y + 1]$ update own $q$
          to $q+$ received $q]$
      else if  $y > N$ and a $Z$ is received from PE$[x, (y + 1)$ mod $2N]$
          then send old $Z$ (if any) to PE on left ]
    {combine records with same $(i, j)$ values}
    if$(\lfloor x$ cosine $+y$ sine$\rfloor = \lfloor x$ cosine $+(y - 1)$ sine$\rfloor)$ and $(0 < y < N)$
      then send $h$ to PE$[x, y - 1]$ and set $i = \phi$
      else if a $q$ value is received set $q = q+$ received $q$;
    send $Z$ to PE$[(w + 1)$ mod $p, y]]$
end
```

Figure 6: MIMD Algorithm

The h values are computed in a pipeline manner. The PEs in row 0 of a $p \times 2N$ subhypercube initiate a record $Z = (i, j$, sine cosine, $q)$ such that $h[i, j] = q$ is the number of pixels on this row that contribute to $h[i, j]$. This is done by first computing i for each pixel in row zero (line 7) for a fixed $j = p/2 - \ell - 2$. Lemma 3.1 is used in lines 22-24 to combine records that represent the same $h[i, j]$ entry. This row of Z records created in row zero moves down the $p \times 2N$ subhypercube one row per iteration (line 25). Lines 10–21 update the row of Z values received. Each such row corresponds to a fixed j. For this j, PE$[w, y]$ determines the h entry $[i', j]$ it is to contribute to (line 13). If this is the same entry as received from PE$[w - 1, y]$ then the two are added together. If $i = \emptyset$ for the received entry, then $[i', j]$ can occupy this Z space. If $i \neq \emptyset$, then from Lemma 3.2 we know that Z can combine only with the new entry $[i', j]$ of PE$[w, y - 1]$.

Following the iteration $\ell = 5p/4 - 1$, the last initiated row (i.e., $j = p/4$) has passed through row $p - 1$ of the $p \times 2N$ subhypercube. At this time, the PEs in row r of the subhypercube contain records with $j = p/4 + r$, $0 \leq r < p/4$. The records in each row may be reordered such that the record in PE$[w, y]$ has $y = i$ by performing a random access write ([NASS81]). Because of the initial ordering of i values in a row, this random access write can be performed in $0(\log N)$ time [RANK 88] rather than in $0(\log^2 N)$ time as required by the more general algorithm of [NASS81].

The phase 2 summing of the $h[i, j]$ values is now easily done in $0(\log N)$ time using window sum. Since the phase 1 algorithm of Figure 3.1 only shifts by 1 along columns and/or rows, each iteration of this algorithm takes only $0(1)$ time. Hence the complexity of the phase 1 algorithm is $0(p)$. The overall time needed to compute H for $p/4 \leq x < p/2$ is therefore $0(p + \log N)$.

The remaining three cases for j are done in a similar way. Actually, the four cases need not be computed independently as suggested above. In particular, all the computation following phase 1 can be done in parallel for all the cases.

4 SIMD Algorithms

We develop two $0(p+\log N)$ SIMD hypercube algorithms. One uses $0(\log N)$ memory per PE while the other uses $0(1)$. The $0(1)$ memory algorithm is slightly more complex than the $0(\log N)$ memory one. Both algorithms are adaptations of our MIMD algorithm. The computations following phase 1 (Figure 6) are easily performed in $0(\log N)$ time on an SIMD hypercube using $0(1)$ memory per PE. So we concentrate on adapting phase 1. The phase 1 algorithm performs $0(p)$ unit shifts along rows and columns of $p \times 2N$ subhypercubes. In an SIMD hypercube, each such row shift takes $0(\log N)$ time while each unit column shift takes $0(\log p)$ time. So a direct simulation of phase 1 takes $0(p \log(Np))$ time.

4.1 $0(\log N)$ Memory per PE

In this case, we divide the $5p/4$ iterations of the *for* loop of Figure 3.1 into blocks of $\log N$ consecutive iterations. In each such block, a Z record initially in PE$[x, y]$ can be augmented by pixel values in PEs $[x+\ell, y-m]$, $0 \leq \ell < \log N$, $-1 \leq m < \log N$. To avoid unit shifts along the rows, each PE$[q, r]$ begins by accumulating the pixel value in PE$[q, r-m]$, $-1 \leq m < \log N$. Now it is necessary to route the Z records only down a column; i.e., a Z record initially in PE $[x, y]$ needs to visit PEs $[x+\ell, y]$, $0 \leq \ell < \log N$. These PEs contain the pixel values needed to update Z to its values following the block of iterations in Figure 3.1. This routing is done using the circulation algorithms in windows of size $\log N$ rather than by unit shifts. The initial pixel accumulation takes $0(\log N)$ time and the circulation and Z updates also take $0(\log N)$ time. Following the circulation, the Z records return to their originating PEs and need to be routed left and down by a distance of $0(\log N)$. This can be accomplished in $0(\log N)$ time on an SIMD hypercube. In this way, we are able to simulate $0(\log N)$ iterations of the MIMD algorithm in $0(\log N)$ time on an SIMD hypercube. Hence the overall asymptotic run time of the SIMD simulation is the same as that of the original MIMD algorithm.

4.2 0(1) Memory per PE

When $\log^2 N/p \leq c$ for some constant, a careful analysis shows that using the strategy employed in the 0(log N) memory algorithm, the memory requirements can be reduced to 0(1). In any $\log N$ block of iterations, two pixels $[x, y]$ and $[w, z]$ contribute to the same Z record only if

$$\lfloor x\cos\theta + y\sin\theta\rfloor = \lfloor w\cos\theta + z\sin\theta\rfloor.$$

Since $w \leq x + \log N - 1$ during the $\log N$ iterations, we get

$$|(\log N - 1)\cos\theta + (z - y)\sin\theta| \leq 1$$

$$\text{or } -\operatorname{cosec}\theta \leq (\log N - 1)\cot\theta + z - y \leq \operatorname{cosec}\theta$$

$$\text{or } -\operatorname{cosec}\theta - (\log N - 1)\cot\theta \leq z - y \leq \operatorname{cosec}\theta - (\log N - 1)\cot\theta.$$

For any fixed $\theta \in [\pi/4, \pi/2]$,

$$z \in [y - (\log N - 1)\cot\theta - \operatorname{cosec}\theta, y - (\log N - 1)\cot\theta + \operatorname{cosec}\theta]$$

$$\text{or } z \in [y - (\log N - 1)\cot\theta - \sqrt{2}, y - (\log N - 1)\cot\theta + \sqrt{2}].$$

There are only a constant number of integers in this range. During a $\log N$ block of iterations, Z records with j value differing by up to $\log N - 1$ may pass through a given PE. This corresponds to a θ variation from θ_1 to θ_2 where $\theta_2 - \theta_1 = \frac{\pi}{p}(\log N - 1)$.

Hence the leftmost column from which a contributing pixel is required has a maximum range of

$$\begin{aligned}
&\operatorname{cosec}\theta_1 + (\log N - 1)\cot\theta_1 - \operatorname{cosec}\theta_2 - (\log N - 1)\cot\theta_2 \\
&\leq \operatorname{cosec}\theta_1 - \operatorname{cosec}\theta_2 + (\log N - 1)(\cot\theta_1 - \cot\theta_2) \\
&\leq \operatorname{cosec}\pi/4 + (\log N - 1)\tfrac{\cos\theta_1\sin\theta_2 - \cos\theta_2\sin\theta_1}{\sin\theta_1\sin\theta_2} \\
&< \operatorname{cosec}\pi/4 + 2(\log N - 1)\sin(\theta_2 - \theta_1) \\
&< \operatorname{cosec}\pi/4 + 2(\log N - 1)(\theta_2 - \theta_1) \\
&= \operatorname{cosec}\tfrac{\pi}{4} + 2(\log N - 1)(\log N - 1)\pi/p \\
&< \operatorname{cosec}\pi/4 + 2\pi c.
\end{aligned}$$

Hence each PE need accumulate only a constant number of pixels from its row rather than the $O(\log N)$ pixels being accumulated in the $O(\log N)$ memory algorithm. This accumulation is done in $O(\log N)$ time. The run time is the same as that of the $O(\log N)$ memory algorithm, but the memory requirements are reduced to $O(1)$.

5 Hough Transform on the NCUBE Hypercube

5.1 NCUBE Architecture

In the previous sections we have developed algorithms to compute the Hough transform on a fine grain hypercube. Such a computer has the property that the cost of interprocessor communication is comparable to that of a basic arithmetic operation. In this section, we shall consider the Hough transform on a hypercube in which interprocessor communication is relatively expensive and the number of processors is small relative to the number of patterns N. In particular, we shall experiment with an NCUBE/7 hypercube which is capable of having up to 128 processors. The NCUBE/7 available to us, however, has only 64 processors. The time to perform a two-byte integer addition on each hypercube processor is 4.3 microseconds, whereas the time to communicate b bytes to a neighbor processor is approximately $447 + 2.4b$ microseconds.

Figure 7 shows the block diagram for the NCUBE/7 hypercube multicomputer.

5.2 Two NCUBE Algorithms

We view the P hypercube nodes as forming rings. Figure 8 shows this ring for the case $P = 8$. For any node i, let left (i) and right (i), respectively, be the node counterclockwise and clockwise from node i. Let logical (i) be the logical index of node i in the ring. The $N \times N$ image array is initially distributed over the nodes with each node getting an $N \times N/qp$ block. Logical node 0 gets the first block, logical node 1 the next block, and so on. Similarly, on completion, the $2\sqrt{2}N \times p$ Hough array H is distributed over the nodes in blocks of size $2\sqrt{2}N \times p/P$. We assume that the number of hypercube nodes P divides the number of angles p as well as the image dimension N. It is further assumed that the thresholding function has already been applied to the pixels and each node has a list of pairs (x, y) such that $I[x, y]$ passes the threshold. We call this list the edge list for the node.

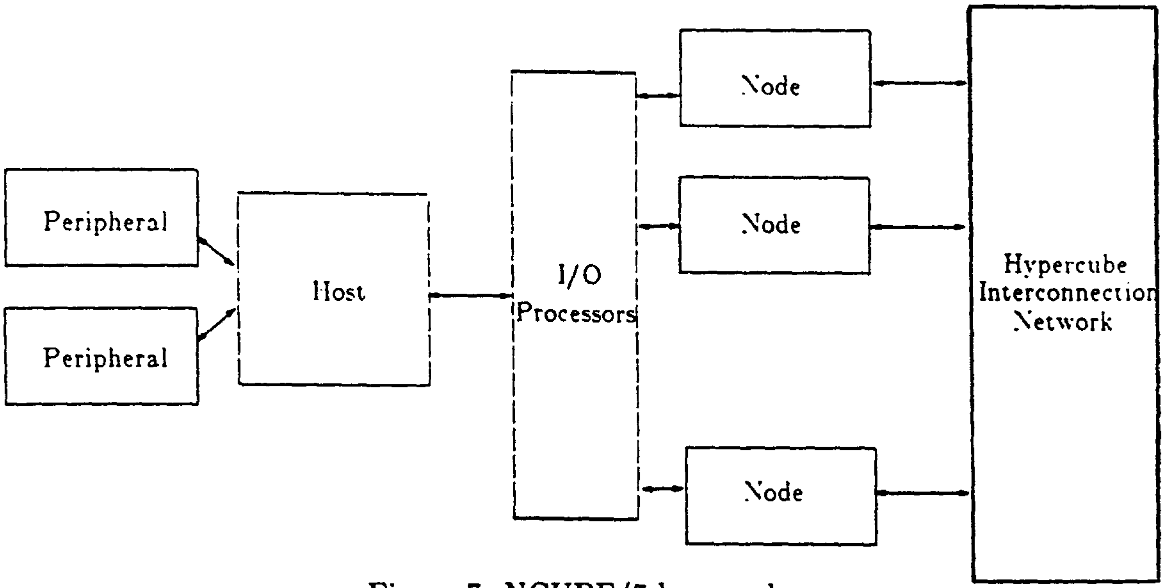


Figure 7: NCUBE/7 hypercube

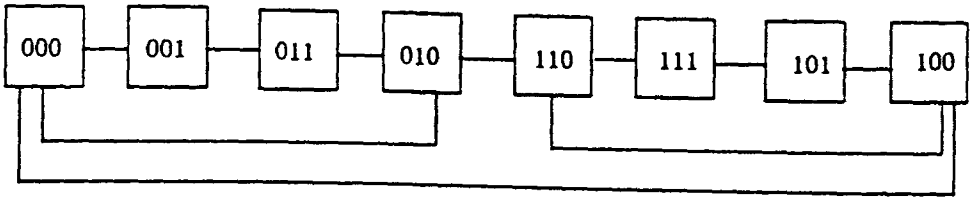


Figure 8: A ring (of size 8) embedded in a hypercube of 8 nodes

```
procedure Update$H$partition ($H$)
        for each $(x, y)$ in edge list $dr$
            for ($j$ := $j$Begin to $j$Begin + size $-1$ do
                $\theta = \frac{\pi}{p}(j+1)$
                $i = x\cos\theta + y\sin\theta$
                increment $H[i,\theta]$ by 1
            end;
        end;
end; {of UpdateHpartition}
$\ell$ := logical index of this node, size:=$p/P$;
$j$ Begin:= size $*\ell$
initialize own $H$ partition to zero;
for $i := 0$ to $P-1$ do
        Update$H$partition;
        send own $H$ partition to node on right;
        receive $H$ partition from node on left;
        $j$ Begin:= ($j$ Begin $-$ size) mod $p$;
end;
```

Figure 9: Non-overlapping algorithm to compute H

Our first algorithm is given in Figure 9. This algorithm is run on each hypercube node. As remarked earlier, each node has an edge list and an H partition.

The H partitions move along the ring one node at a time. When an H partition reaches any node, the edge list of that node is used to update it, accounting for all contributions these edges make to this H partition. Procedure Update$H$Partition does precisely this. *jBegin* is the j value corresponding to the first angle (column) in the H partition currently in the node. *size* $= p/P$ is the number of columns in an H partition.

In the algorithm of Figure 9 no attempt is made to overlap computation with communication. Following the send of an H partition to its right neighbor, the node is idle until the receive of the H partition from its left neighbor is complete. Figure 10 shows the activity of a node as a function of time.

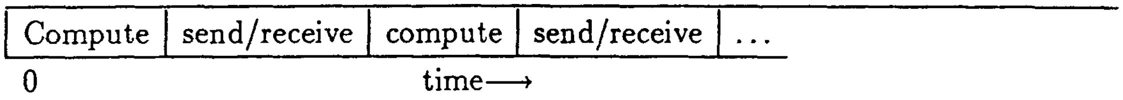


Figure 10. Nonoverlapping Algorithm to Compute H

During the compute phase, an H partition is updated. Let t_c be the time needed to do this. Let t_t be the time for an H partition to travel from a sending node to its destination node. So t_t is the elapsed time between the initiation of the transfer and the receipt of the partition. The time required by the nonoverlapping algorithm of Figure 9 is $P(t_c + t_t)$.

```
$\ell$ := logical index of this node; size = $p/P$;
$jBegin := size*\ell$;
for $i := 0$ to $P-1$ do

  if $i = 0$ then [initialize own $H$ partition to zero
                  Update $H$ Partition ($H$)]
          else [initialize $T$ to zero
                  Update $H$ Partition ($T$)
                  Receive $H$ Partition from left ($\ell$)
                  $H := H + T$]
  send $H$ to right ($\ell$);
  $jBegin$ := ($jBegin$−size) mod $p$
end;
```

Figure 11. Overlapping Algorithm for H

Our second algorithm, (Figure 11), attempts to overlap as much of the transmission time t_t with computation. This, unfortunately, results in an increase in the computation time as some additional work is to be done. At the end of each iteration of the *for* loop, the H partition in a node ℓ is sent to the node on its right. The next iteration proceeds while the H partition is in transit. For this, a temporary space T of the same size as H is used to accumulate the contribution of this node's edge list to the H partition it has yet to receive from its left neighbor. Following this computation, the received H portion and T are added as the resulting H partition transmitted to the right.

Relative to the nonoverlapping algorithm, the overlapping algorithm does $P-1$ initializations of T and executions of $H := H+T$ extra computational work. Let t_{init} be the time to initialize T and t_{add} the time to execute $H := H+T$. If $t_t \leq t_{init} + t_c$, the time diagram has the form shown in Figure 12(a). The overall time for the algorithm is $Pt_c + (P-1)(t_{init} + t_{add}) + t_t$ when $t_t \leq t_{init} + t_c$. So if $t_{init} + t_{add} < t_t$, the overlapping algorithm will outperform the nonoverlapping algorithm.

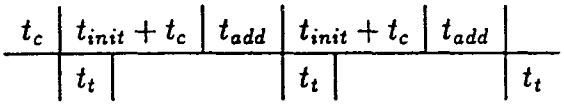


Figure 12 (a) $t_t \leq t_{init} + t_c$

When $t_t = t_{init} + t_c + \Delta t, \Delta t > 0$, the time diagram is as in Figure 12 (b). In this case, the algorithm run time is $t_c+(P-1)t_{add}+Pt_t = Pt_c+(P-1)(t_{init}+t_{add}+\Delta)+t_t$. For the overlapping algorithm to outperform the nonoverlapping algorithm, we need $t_{init} + t_{add} + \Delta < t_t$.

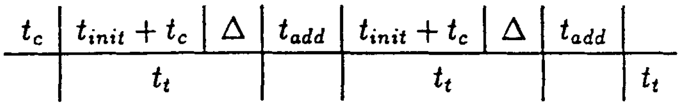


Figure 12 (b) $t_t = t_{init} + t_c + \Delta,\ \Delta > 0$

5.3 Load Balancing

The preceding analysis is somewhat idealistic as it assumes that t_c is the same in each node. Actually, the size of the edge list in each node is different and this difference significantly impacts the performance of the algorithm. The node with the maximum number of edges becomes a bottleneck. To reduce the run time, one may attempt to obtain an equal or near equal distribution of the edges over the P nodes. Note that even though the image matrix I is equally distributed over the nodes, the edge lists may not be, as a different number of pixels in each I partition will pass the threshold. We shall use the term *load* to refer to the number of pixels in a node that passes this threshold. I.e., load is the size of the nodes' edge list. Two heuristics to balance the load are given in Figures 13 and 14.

In both, load balancing is accomplished by averaging over the load in processors that are directly connected. The variables used have the following significance:

MyLoad=current load in the node processor
HisLoad=load in a directly connected node processor
MyLoadSize=size of the load in the node processor
HisLoadSize=size of the load in a directly connected node processor
avg=average size of the load of the two processors

```
procedure LoadBalance1();
    for i := 0 to CubeSize do
        Send MyLoad to neighbor processor along dimension i;
        Receive HisLoad from neighbor processor along dimension i
        and append to Myload;
        avg=(MyLoadSize+HisLoadSize+1)/2;
        if (MyLoadSize > Avg) MyLoadSize=Avg;
        else if (HisloadSize>Avg) MyLoadSize+=HisLoadSize - Avg;
    end;
end;
```

Figure 13. Load Balancing (Heuristic 1)

```
procedure LoadBalance2();

    for $i$ := 0 to CubeSize do
      Send MyLoadSize to neighbor processor along dimension $i$;
      Receive HisLoadSize from neighbor processor along
      dimension $i$;
      avg=(MyLoadSize+HisLoadSize+1)/2;
      if (MyLoadSize > Avg)[
        Send extra load (MyLoadSize−Avg) to neighbor
        processor along dimension $i$;
        MyLoadSize = Avg; ]
      else if (HisLoadSize>Avg)[
        Receive extra load (Avg−HisLoadSize) from neighbor
        processor along dimension $i$
        MyLoadSize+ =HisLoadSize−Avg;]
    end;
end;
```

Figure 14. Load Balancing (Heuristic 2)

The only difference between the two variations is that in the first one a processor transmits its entire work load (including the necessary data) to its neighbor processor, while in the second variation only the amount in excess of the average is transmitted. However, in order to achieve this reduction in load transmission, it is necessary to first determine how much of the load is to be transmitted. This requires an initial exchange of the load size. Hence variation 2 requires twice as many message transmissions. Each message of variation 2 is potentially shorter than each message transmitted by variation 1. We expect variation 1 to be faster than variation 2 when the number of bytes in MyLoad and HisLoad is relatively small and the time to set up a data transmission relatively large. Otherwise, variation 2 is expected to require less time.

5.4 Experimental Results

The nonoverlapping and overlapping algorithms of Section 5.2, as well as the load balancing heuristics of Section 5.3 were programmed in C and run on an NCUBE/7 hypercube with 64 nodes. We experimentec with randomly generated images of size $N \times N$ for $N = 32$, 64, 128, 256, and 512. The percentage of pixels in an $N \times N$ image that passed the threshold was fixed at 5%, 10%, or 20%. The number of edge pixels in each nodes I partition was determined using a truncated normal distribution with variance being one of 4%, 10%, and 64% of the mean. In all cases, we set $p = 180$.

Preliminary experiments indicated that the run time of our two load-balancing heuristics was approximately the same, with the second heuristic having a slight edge. Furthermore, the time to load balance is less than 2% of the overall run time (load balance followed by Hough transform computation). The run time of the nonoverlapping algorithm, both with and without load balancing, is given in Figures 16, 17, and 18 for the cases of $P = 4$, 16, and 64, respectively. We see that as the load variance increases from 4% to 64%, the run time of the nonoverlapping algorithm without load balancing increases significantly. In fact, it almost doubles. With load balancing, however, the run time is quite stable. Furthermore, it is always less than the run time for 4% variance without load balancing. When the variance in load is 64%, load balancing results in a 25% to 53% reduction in run time!

Note that the average load per node when $P = 4$ and $N = 128$ is the same as when $P = 16$ and $N = 256$ and when $P = 64$ and $N = 512$. From Figures 16, 17, and 18 we see that run time remains virtually unchanged as P increases, provided the load per node is unchanged. Hence the algorithm scales well.

The run times for the overlapping algorithm with load balancing are given in Figure 19. These times are generally slightly larger than those for the nonoverlapping algorithm with load balancing. So, the computational overhead introduced by the overlapping algorithm more or less balances the positive effects of overlapping computation and communication.

For comparison purposes, the run times on a single hypercube node are given in Figure 15 for the cases $N = 16$, 32 and 64. The case $N = 128$ could not be run for lack of sufficient memory.

Speedup and efficiency are common measures of the goodness of a parallel algorithm. Speedup is defined as:

$$S_p = \frac{\text{run time}}{\text{time taken by a uniprocessor}}$$

while efficiency, E_p, is defined as:

$$E_p = \frac{S_p}{P}.$$

Image Size	% edges	Time in Seconds
16 × 16	5	0.3005
	10	0.5636
	20	1.1016
32 × 32	5	1.1597
	10	2.2209
	20	4.3527
64 × 64	5	4.4399
	10	8.7194
	20	17.2660

number of nodes = 1

Figure 15

Image Size	%	No Load Balancing			Load Balance 2		
		Variance					
	edges	4%	16%	64%	4%	16%	64%
32×32	5	0.2802	0.3138	0.3940	0.2819	0.2785	0.2804
	10	0.5627	0.6035	1.1563	0.5531	0.5527	0.5527
	20	1.1439	1.3364	1.7874	1.0976	1.0956	1.0967
64×64	5	1.1465	1.3044	1.7575	1.1202	1.1187	1.1176
	10	2.2428	2.4485	3.4152	2.1878	2.1853	2.1818
	20	4.4974	4.7970	8.0548	4.3171	4.3238	4.3190
128×128	5	4.4966	5.0359	7.8626	4.3605	4.3550	4.3564
	10	8.9968	10.0017	15.5813	8.6474	8.6393	8.6423
	20	18.0087	19.1119	31.7456	17.2247	17.2349	17.2108

Number of nodes=4, no overlap

Figure 16

Image Size	%	No Load Balancing			Load Balance 2		
		Variance					
	edges	4%	16%	64%	4%	16%	64%
64 × 64	5	0.2964	0.3494	0.5622	0.2981	0.3012	0.2922
	10	0.5949	0.6803	1.1556	0.5927	0.5830	0.5712
	20	1.1827	1.4140	2.0260	1.1615	1.1574	1.1313
128 × 128	5	1.2088	1.4113	2.2415	1.1915	1.1798	1.1570
	10	2.3558	2.7429	5.3075	2.3256	2.3065	2.2469
	20	4.6616	5.3293	9.0813	4.5909	4.5600	4.4518
256 × 250	5	4.6854	5.6429	9.3724	4.6283	4.5810	4.4624
	10	9.3130	11.0024	18.0237	9.1721	9.1270	8.8296
	20	18.4712	21.4809	33.9781	18.2738	18.1359	17.6917

Number of nodes=16, no overlap

Figure 17

Image Size	%	No Load Balancing			Load Balance 2		
		Variance					
	edges	4%	16%	64%	4%	16%	64%
128 × 128	5	0.3462	0.4200	0.6449	0.3416	0.3481	0.3400
	10	0.6512	0.7735	1.3975	0.6313	0.6315	0.6232
	20	1.2692	1.5239	2.8156	1.2051	1.2062	1.1960
256 × 256	5	1.2638	1.5371	2.7062	1.2291	1.2229	1.2057
	10	2.4770	2.9324	5.1395	2.3543	2.3476	2.3288
	20	4.9057	6.0051	11.4614	4.6170	4.6020	4.5470
512 × 512	5	4.9077	5.8094	10.8207	4.6485	4.6232	4.5784
	10	9.7492	11.7256	20.7631	9.1908	9.1611	9.0623
	20	19.3672	23.9020	38.0617	18.2782	18.2166	18.0306

Number of nodes=64, no overlap

Figure 18

Block	%	$P = 4$		
Size	edges	4%	16%	64%
32	5	0.3704	0.3689	0.3708
×	10	0.6424	0.6425	0.6925
32	20	1.1862	1.1851	1.1857
64	5	1.3030	1.3011	1.3009
×	10	2.3686	2.3680	2.3614
64	20	4.4927	4.4967	4.4956
128	5	4.7045	4.6999	4.7019
×	10	8.9787	8.9760	8.9835
128	20	17.5374	17.5426	17.5304

Figure 19

Block	%	$P = 16$			$P = 64$		
Size	edges	4%	16%	64%	4%	16%	64%
32	5	0.4081	0.4152	0.4094	0.4578	0.4619	0.4597
×	10	0.6843	0.6806	0.6810	0.7328	0.7349	0.7303
32	20	1.2211	1.2263	1.2275	1.2705	1.2752	1.2743
64	5	1.3675	1.3682	1.3685	1.4292	1.4214	1.4173
×	10	2.4359	2.4324	2.4307	2.4802	2.4848	2.4860
64	20	4.5572	4.5603	4.5570	4.6034	4.6039	4.6100
128	5	4.8167	4.8148	4.8151	4.8761	4.8691	4.8718
×	10	9.0806	9.0850	9.0970	9.1464	9.1467	9.1344
128	20	17.6483	17.6388	17.6390	17.6879	17.6730	17.6907

Overlap Communication and Computation

Figure 20

edges	No. of	Image= 64 × 64		Image= 128 × 128	
	Nodes	Time	Speedup	Time	Est.Speedup
5	1	3.8603	1.0000		
	4	0.9787	3.9440	0.3964	3.9440
	16	0.2844	13.5728	1.0734	14.5640
	64	0.1551	24.8754	6.3414	45.7945
10	1	7.6151	1.0000		
	4	1.91169	3.9724	8.3301	3.9724
	16	0.5263	14.4682	2.2187	14.9288
	64	0.2046	37.2515	0.6278	52.7058
20	1	15.6470	1.0000		
	4	3.9246	3.9868	17.0167	3.9868
	16	1.0529	14.8604	4.4732	15.1660
	64	0.3374	46.3741	1.1777	56.6400

No overlap between communication/computation
Variance of edge = 64%
Figure 21

Figure 21 gives the speedup and efficiency figure achieved by our nonoverlapping algorithm with load balancing for the cases: variance= 64%, %edges=20, and N = 64 and 128. These are plotted in Figures 21 and 22, respectively.

6 Conclusions

We have developed efficient hypercube algorithms for the Hough transform problem. The fine grain algorithms are optimal and the algorithms for a medium grain hypercube exhibit near-optimal speedups when load balancing is done.

7 Bibliography

[**CHAN86**] T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor," *IEEE Transactions on Computers*, vol. C-35, pp. 969–977, Nov. 1986.

[**CHAN87**] S. Chandran and L. Davis, "The Hough Transform on the butterfly and the ncube," *University of Maryland Technical Report*, 1987.

[**CYPH87**] R. E. Cypher, J. L. C. Sanz, "The Hough Transform has $O(N)$ complexity on SIMD $N \times N$ Mesh Array Architectures," *Proc. of IEEE CAPAMI Workshop*, 1987.

[**DEKE81**] E. Dekel, D. Nassimi and S. Sahni, "Parallel matrix and graph algorithms," *SIAM Journal on Computing*, pp. 657–675, 1981.

[**FISH87**] A. Fisher and P. Highnam, "Computing the Hough transform on a scan line array processor," *Proc. of IEEE CAPAMI Workshop*, 1987, pp. 83–87.

[**GUER87**]C. Guerra and S. Hambrusch, "Parallel algorithms for line detection on a mesh," *Proc. of IEEE CAPAMI Workshop*, 1987, pp. 99–106.

[**HORO85**] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, 1985.

[**IBRA86**] H. Ibrahim, J. Kender, D. E. Shaw, "On the Application of Massively Parallel SIMD Tree Machines to Certain Intermediate Level Vision Tasks," *CVGIP 36*, pp. 53–75, 1986.

[**NASS81**] D. Nassami and S. Sahni, "Data Broadcasting in SIMD Computers," *IEEE Transactions on Computers*, No. 2, vol. C–301, pp. 101–107, Feb. 1981.

[**PRAS87**] V. K. Prasanna Kumar and V. Krishnan, "Efficient Image Template Matching on SIMD Hypercube Machines," *Proceedings of 1987 International Conference on Parallel Processing*, pp. 765–771.

[ROSE82] A. Rosenfeld and A. C. Kak, "Digital Picture Processing," Academic Press, 1982.

[ROSE88] A. Rosenfeld, J. Ornelas and Y. Hung, "Hough Transform Algorithms for Mesh-connected SIMD Parallel Processors," *CVGIP 41*, No. 3, pp. 293–305, 1988.

[RANK88] S. Ranka and S. Sahni, "Image Template Matching on an SIMD hypercube multicomputer," *Proceedings of 1988 International Conference on Parallel Processing*, 1988.

[THOM77] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Communications of the ACM*, pp. 263–271, 1977.