Image Transformations on Hypercube and Mesh Multicomputers

Sanjay Ranka
Syracuse University

and

Sartaj Sahni
University of Minnesota

Abstract

Efficient hypercube and mesh algorithms are developed for the following image transformations: shrinking, expanding, translation, rotation, and scaling. A 2^k*–step* shrinking and expanding of a gray scale $N \times N$ image can be done in $O(k)$ time on an N^2 processor MIMD hypercube, in $O(\log N)$ time on an SIMD hypercube, and in $O(2^k)$ time on an $N \times N$ mesh. Translation, rotation, and scaling of an $N \times N$ image take $O(\log N)$ time on an N^2 processor hypercube and $O(N)$ time on an $N \times N$ mesh..

1 Introduction

Since transformations on two dimensional images are very compute intensive, several authors have developed parallel algorithms for these. For example, Rosenfeld, [11], develops pyramid algorithms for shrinking and expanding. Using his algorithms, a 2^k step resampled shrinking and expanding of an $N \times N$ image can be performed in $O(k)$ time on a pyramid with an $N \times N$ base and height k. For unresampled shrinking and expanding, [11] develops an $O(k^2)$ algorithm for a one dimensional binary image. The generalization of this algorithm to two dimensional images results in a pyramid algorithm of complexity $O(2^k)$. The unresampled algorithm of [11] does not generalize to the case of grayscale images. In this chapter, we develop hypercube algorithms for unresampled shrinking/expanding. Our algorithms may be applied to both binary as well as grayscale images. Our algorithm for a 2^k*–step* shrinking or expanding on an $N \times N$ image takes $O(k)$ time on an N^2 processor MIMD hypercube, $O(\log N)$ time on an N^2 processor SIMD hypercube, and $O(2^k)$ time on an $N \times N$ SIMD and MIMD mesh.

Lee, Yalamanchali, and Aggarwal, [3], develop parallel algorithms for image translation, rotation and scaling. Their algorithms are for a mesh connected multicomputer. Their algorithms are able to perform the above operations on an $N \times N$ binary image in $O(N)$ time, when an $N \times N$ processor mesh is available. We show how these operations can be performed for a grayscale image in $O(\log N)$ time using an N^2 processor hypercube. Additionally, we develop mesh algorithms for translation, rotation, and scaling that complete in $O(N)$ time and use only $O(1)$ memory per processor. While these operations can be performed in these time and space bounds using random access writes [4], our algorithms are elegant and have smaller constant factors associated with the complexity.

In the next section we describe our hypercube and mesh models and some basic data

1. Throughout this chapter, we assume N is a power of 2.

movement operations. In section 3, our algorithms for shrinking and expanding are developed. Algorithms for translation, rotation, and scaling are presented in sections 4, 5, and 6 respectively.

2 Multicomputer Model

The important features of an SIMD multicomputer and the programming notation we use are:

1. There are P processing elements connected together via an interconnection network. Different interconnection networks lead to different SIMD architectures. Each processing element (PE) has a unique index in the range $[0, P-1]$. 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

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

   $(i_0 = 1)$ 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. So, the above statement is equivalent to the statement:

   $$A := A + 1, \quad (i_0 = 1)$$

3 We shall consider the following interconnection networks:

   a) **Hypercube**: A p dimensional hypercube network connects $P = 2^p$ PEs (?F{2} (a)). Let $i_{p-1}i_{p-2} \cdots i_0$ be the binary representation of the PE index i. Let $\overline{i_k}$ denote the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit. I.e., processor $i_{p-1}i_{p-2} \cdots i_0$ is connected to processors $i_{p-1} \cdots \overline{i_k} \cdots i_0$, $0 \le k \le p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

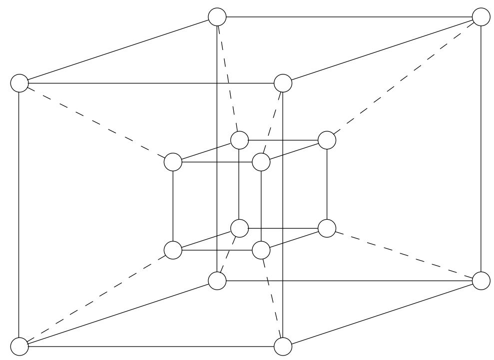
   b) **Mesh**: A $P = N \times N$ mesh connects N^2 PEs that are logically arranged as a two dimensional array (?F{2} (b)). Each PE has a unique index in the range $(0..N-1, 0..N-1)$. PE(i, j) is connected to PE$((i-1) \; mod \; N, j)$, PE$((i+1) \; mod \; N, j)$, PE$(i, (j-1) \; mod \; N)$ and PE$(i, (j+1) \; mod \; N)$. Sometimes we will use a one dimensional indexing of the mesh. This is obtained using the standard row major mapping in which (i, j) is mapped $iN + j$.

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

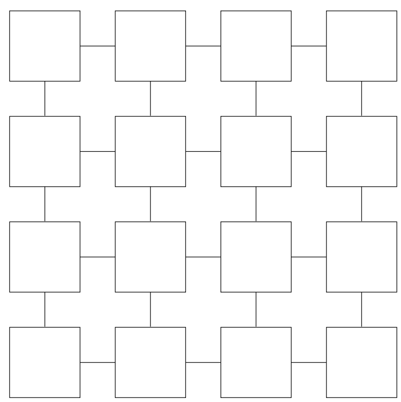
   $$B(i^{(2)}) \leftarrow B(i), \quad (i_2 = 0)$$

   on a hypercube is executed only by those processors whose bit 2 is equal to 0. These processors transmit their B register data to the corresponding processors whose bit 2 is 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



(a) 16 processor hypercube



(b) A 16 node mesh
End around connections are not shown

?F{2} Interconnection networks

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 $i^{(b)}$. For example, on a hypercube, 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 multicomputers differ from those of SIMD multicomputers 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, in an MIMD hypercube, PE i may transfer data to PE $i^{(b)}$, while PE j simultaneously transfers data to PE $j^{(a)}$, $a \neq b$.

3 Image Mapping

In the case of a mesh, the pixel in position $[i, j]$ of the image is mapped to processor (i, j) of the mesh. For a hypercube, a two dimesional grid view is needed. ?F{p3}(a) gives a two dimensional grid interpretation of a dimension 4 hypercube. This is the binary reflected gray code mapping of [1]. An i bit binary gray code S^i is defined recursively as below:

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

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.

0000	0001	0011	0010
0100	0101	0111	0110
1100	1101	1111	1110
1000	1001	1011	1010

(a) Gray code mapping

0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111

(b) Row major mapping

?F{p3} A 16 PE hypercube viewed as an 4×4 grid

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 snake like row major order [12]. This mapping has the property that grid elements that are neighbors are assigned to neighboring hypercube nodes.

?F{p3}(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 [2]. I.e, for an $N \times N$ grid, the PE at position (i, j) has index $iN + j$. Using the mapping, a two dimensional image grid $I[0..N, 0..N]$ 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 hypercube algorithms and the row major mapping for all SIMD hypercube algorithms.

4 Basic Data Manipulation Operations

4.1 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 $A(qW + (j-i) \; mod \; W)$, $0 \le q < (P/W)$. $SHIFT(A, i, W)$ on an SIMD hypercube computer can be performed in $2\log W$ unit routes [6]. A minor modification of the algorithm given in [6] performs $i = 2^m$ shifts in $2\log(W/i)$ unit routes [7]. 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 $A(qW + j - i)$, so long as $0 \le j - i < W$ and by 0 otherwise. This change does not increase the number of unit routes. The end off shift will be denoted $ESHIFT(A, i, W)$. Both *SHIFT* and *ESHIFT* can be done, in a straightforward way, in $|i|$ unit routes on an SIMD mesh.

4.2 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 hypercube [8]. On an MIMD mesh, the operation is easily performed with $|i|$ unit routes. 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.

4.3 Row and Column Reordering

These are special cases of the random access write (RAW) operation defined in [4]. We assume an $N \times N$ array logical view of an N^2 PE hypercube. In a row reordering the destination processor, $dest(p)$, for data in any PE is another PE in the same row. The $dest()$ values in each row of the $N \times N$ processor array are either nondecreasing left to right for all rows or nonincreasing left to right for all rows. Because of this monotonicity of the *dest* values, the required reordering can be done in in $O(d \log N)$ where d is the maximum number of processors in any row that have the same *dest*. In case only one of the many data destined to the same processor is to survive, the time can be reduced to $O(\log N)$. This reduction requires that the surviving data be selected by some associative operation like min or max. The algorithm for this is obtained from the RAW algorithm of [4] by omitting the sort step.

Column reordering is the column analog of row reordering. It is performed in an analogous manner. Both operations take $O(N)$ time on a mesh.

5 Shrinking And Expanding

Let $I[0..N-1, 0..N-1]$ be an $N \times N$ image. The *neighborhood* of the image point $[i, j]$ is defined to be the set:

$nbd(i, j) = \{ [u, v] \mid 0 \le u < N,\ 0 \le v < N,\ \max\{ |u-i|, |v-i| \} \le 1 \}$

The q*-step shrinking* of I is defined in [10] and [11] to be the $N \times N$ image S^q such that:

$S^q[i, j] = \min_{[u, v] \in nbd(i, j)} \{ I[u, v] \},\ q = 1,\ 0 \le i < N,\ 0 \le j < N$

$S^q[i, j] = \min_{[u, v] \in nbd(i, j)} \{ S^{q-1}[u, v] \},\ q > 1,\ 0 \le i < N,\ 0 \le j < N$

Similarly, the q*-step expansion* of I is defined to be an $N \times N$ image E^q such that:

$E^q[i, j] = \max_{[u, v] \in nbd(i, j)} \{ I[u, v] \},\ q = 1,\ 0 \le i < N,\ 0 \le j < N$

$E^q[i, j] = \max_{[u, v] \in nbd(i, j)} \{ E^{q-1}[u, v] \},\ q > 1,\ 0 \le i < N,\ 0 \le j < N$

When the images are binary, the min and max operators in the above definitions may be replaced by *and* and *or* respectively. Let $B_{2q+1}[i, j]$ denote the block of pixels:

$\{ [u, v] \mid 0 \le u < N,\ 0 \le v < N,\ \max\{ |u-i|, |v-i| \} \le q \}$

Then $nbd(i, j) = B_3[i, j]$. In [11], it is shown that:

$$S^q[i, j] = \min_{[u, v] \in B_{2q+1}(i, j)} I[u, v],\ 0 \le i < N,\ 0 \le j < N \qquad (1)$$

$$E^q[i, j] = \max_{[u, v] \in B_{2q+1}(i, j)} I[u, v],\ 0 \le i < N,\ 0 \le j < N$$

Our remaining discussion of shrinking and expanding will explicitly consider shrinking only. Our algorithms for shrinking can be easily transformed to expanding algorithms of the same complexity. This transformation simply requires the replacement of every min by a max and a change in the *ESHIFT* fill in from ∞ to $-\infty$. In the case of binary images the min and max operators may be replaced by *and* and *or* respectively and the *ESHIFT* fill in of ∞ and $-\infty$ by 1 and 0 respectively.

Let $R^q[i, j]$ be defined as below:

$$R^q[i, j] = \min_{[i, v] \in B_{2q+1}(i, j)} I[i, v],\ 0 \le i < N,\ 0 \le j < N \qquad (2)$$

From (1), it follows that

$$S^q[i, j] = \min_{[u, j] \in B_{2q+1}(i, j)} R^q[u, j],\ 0 \le i < N,\ 0 \le j < N \qquad (3)$$

When an $N \times N$ image is mapped onto an $N \times N$ MIMD or SIMD hypercube or mesh using the mappings of Section 2, the rows and columns of the mappings are symmetric. Consequently, the algorithms to compute R^q and S^q from (2) and (3) are very similar. Hence, in the sequel we consider the computation of R^q only. In keeping with the development of [11], we assume $q = 2^k$.

5.1 MIMD Hypercube Shrinking

On an MIMD hypercube R^q for $q = 2^k$ may be computed using the algorithm of ?P{mimdShrink}. The computation of R is done in two stages. These are obtained by decomposing (2) into

$$left^q[i, j] = \min_{\substack{[i, v] \in B_{2q+1}[i, j] \\ v \le i}} \{I[i, v]\}, \ 0 \le i < N, \ 0 \le j < N$$

$$right^q[i, j] = \min_{\substack{[i, v] \in B_{2q+1}[i, j] \\ v \ge i}} \{I[i, v]\}, \ 0 \le i < N, \ 0 \le j < N$$

$$R^q[i, j] = \min\{ left^q[i, j], right^q[i, j] \} \ 0 \le i < N, \ 0 \le j < N$$

```
procedure MIMDShrink;
{Compute R^q for q = 2^k on an MIMD hypercube}
begin
  {compute min of the left 2^k pixels on the same row}
  {MIMDEShift does an ∞ fill instead of a 0 fill}
  left(p) := I(p);
  for i :=0 to k − 1 do
  begin
    C(p) := left(p);
    MIMDEShift(C, 2^i, N);
    left(p) := min {left(p), C(p)};
  end
  C(p) := I(p);
  MIMDEShift(C, 2^k, N);
  left(p) := min {left(p), C(p)};

  {compute min of the right 2^k pixels on the same row}
  right(p) := I(p);
  for i :=0 to k − 1 do
  begin
    C(p) := right(p);
    MIMDEShift(C, −2^i, N);
    right(p) := min {right(p), C(p)};
  end
  C(p) := I(p);
  MIMDEShift(C, −2^k, N);
  right(p) := min {right(p), C(p)};

  R(p) := min {left(p), right(p)}
end;
```

?P{mimdShrink} MIMD computation of R^q

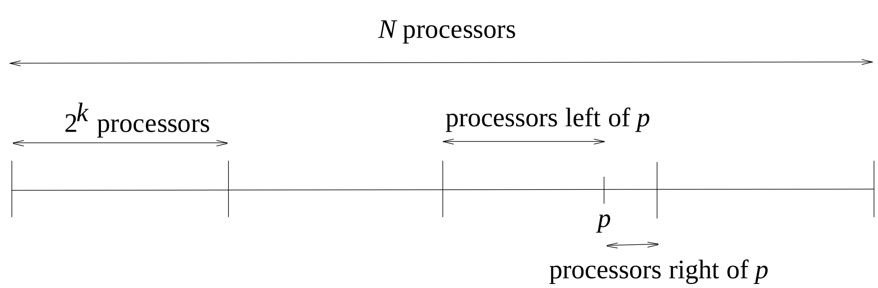
One may verify that following the first **for** loop iteration with $i = a$, $left(p)$ is the min of the pixel values in the left 2^a processors and that in its own I register, $0 \le a < k$. To complete the computation of *left* (p) we need also to consider the pixel value 2^k units to the left and on the same image row. This is done by a rightward shift of 2^k. The shift is done by rows (i.e., blocks of size N) with a fill in of ∞. A similar argument establishes the correctness of the second stage computation of *right*.

Since a power of 2 MIMD shift takes O(1) time, it follows that the complexity of procedure *MIMDShrink* is $O(k)$. Once R^q, $q = 2^k$, has been computed, S^q may be computed using a similar algorithm.

5.2 SIMD Hypercube Shrinking

Since a shift of 2^i in a window of size N takes $O(\log(N/2^i))$ time on an SIMD hypercube, a simple adaptation of *MIMDShrink* to SIMD hypercubes will result in an algorithm whose complexity is $O(k\log N)$. We can do better than this using a different strategy.

The $N \times N$ image is mapped onto the $N \times N$ hypercube using the row major mapping. R^q for $q = 2^k$ may be computed by considering the N processors that represent a row of the image as comprised of several blocks of size 2^k each (see ?F{block}).



?F{block} 2^k blocks of processors

Each processor p computes

left(p) = minimum of pixel values to the left of p but within the same 2^k block

right(p) = minimum of pixel values to the right of p but within the same 2^k block

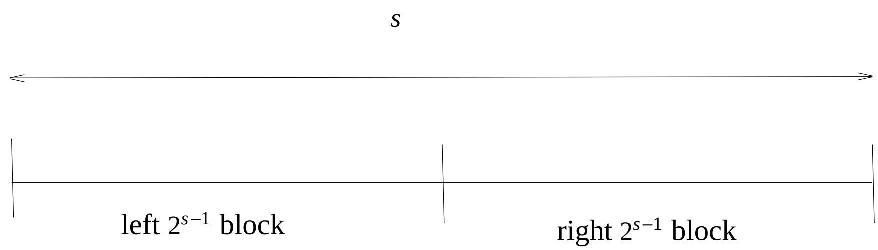
Now, $R^q(p)$ is the minimum of:

(a) I(p)
(b) *left*(p)
(c) *right*(p)
(d) *left*$(p+q)$ provided $p+q$ is in the same row
(e) *right*$(p-q)$ provided $p-q$ is in the same row

Note that this is true even if q is not a power of 2 and we use $k = \lfloor \log_2 q \rfloor$ in the definition of *left* and *right*. *left(p)* and *right(p)* for 2^k blocks may be computed by first computing these for

2^0 blocks, then for 2^1 blocks, then 2^2 blocks and so on. Let *whole* (p) be the minimum of all pixels in the block that currently contains PE p. For 2^0 blocks, we have

$$left(p) = right(p) = \infty;\ whole(p) = I(p)$$



?F{sblock} A 2^s block of processors

Each 2^s block for $s > 0$ consists of two 2^{s-1} blocks as shown in ?F{sblock}. One is the left 2^{s-1} block and the other the right 2^{s-1} block. The PEs in the left 2^{s-1} block have bit $s-1 = 0$ while those in the right one have bit $s-1 = 1$. Let us use a superscript to denote block size. So, $left^{\,s}(p)$ denotes *left(p)* when the block size is 2^s. We see that when p is in the left 2^{s-1} block,

$$left^{\,s}(p) = left^{\,s-1}(p)$$

$$right^{\,s}(p) = \min\{right^{\,s-1}(p),\ whole^{\,s-1}(p + 2^{s-1})\}$$

$$whole^{\,s}(p) = \min\{whole^{\,s-1}(p),\ whole^{\,s-1}(p + 2^{s-1})\}$$

and when p is in the right 2^{s-1} block,

$$left^{\,s}(p) = \min\{left^{\,s-1}(p),\ whole^{\,s-1}(p - 2^{s-1})\}$$

$$right^{\,s}(p) = right^{\,s-1}(p)$$

$$whole^{\,s}(p) = \min\{whole^{\,s-1}(p),\ whole^{\,s-1}(p - 2^{s-1})\}$$

?P{simdShrink} implements the strategy just developed. Its complexity is $O(\log N)$. The algorithm can also be used when q is not a power of 2 by simply defining $k = \lfloor \log q \rfloor$. The complexity remains $O(\log N)$.

5.3 Shrinking on SIMD and MIMD Meshes

The operations required to shrink on a mesh are readily performed in $O(2^k)$ time.

```
procedure SIMDShrink;
{Compute R^q for q = 2^k on an SIMD hypercube}
begin
   {initialize for 2^0 blocks}
   whole(p) := I(p);
   left(p) := ∞;
   right(p) := ∞;

   {compute for 2^(i+1) blocks}
   for i :=0 to k − 1 do
   begin
      C(p) := whole(p);
      C(p) ← C(p^(i));
      left(p) := min {left(p), C(p)}; (p^(i) = 1)
      right(p) := min {right(p), C(p)}; (p^(i) = 0)
      whole(p) := min {whole(p), C(p)};
   end

   R(p) :=min { I(p), left(p), right(p)}
   SIMDEShift(left, −q, N);
   SIMDEShift(right, q, N);
   R(p) :=min { R(p), left(p), right(p)}
end; {of SIMDShrink}
```

?P{simdShrink} SIMD computation of R^q

6 Translation

This operation requires moving the pixel at position $[i, j]$ to the position $[i + a, j + b]$, $0 \le i < N$, $0 \le j < N$ where a and b are given and assumed to be in the range $0 \le a, b \le N$. Translation may call for image wraparound in case $i + a \ge N$ or $j + b \ge N$. Alternatively pixels that get moved to a position $[c, d]$ with either $c \ge N$ or $d \ge N$ are discarded and pixel positions $[i, j]$ with $i < a$ or $j < b$ get filled with zeroes. Regardless of which alternative is used, image translation can be done by first shifting by a along rows (circular shift for wraparound or zero fill right shift for no wraparound) and then shifting by b along rows. Unless a and b are powers of 2, the time complexity is $O(\log N)$ on both an SIMD and an MIMD hypercube. When a and b are powers of 2, the translation takes $O(1)$ time on an MIMD hypercube. The shifting needed for the translation takes $O(|a| + |b|)$ time.

7 Image Rotation

The $N \times N$ image I is to be rotated θ^0 about the point $[a, b]$ where a and b are integers in the range $[0, N-1]$. Following the rotation, *pixel* $[i, j]$ of I will be at position $[i', j']$ where i' and j' are given by [9]:

$i' = \lceil (i-a)\cos\theta - (j-b)\sin\theta + a \rceil$

$j' = \lceil (i-a)\sin\theta + (j-b)\cos\theta + b \rceil$

The equations for i' and j', may be simplified to:

$i' = \lceil i\cos\theta - j\sin\theta + A \rceil$ (4)

$j' = \lceil i\sin\theta + j\cos\theta + B \rceil$

where $A = a(1-\cos\theta) + b\sin\theta$

and $B = b(1-\cos\theta) - a\sin\theta$

We first consider rotations of $\theta = 180^0, 90^0, -90^0$, and $|\theta| \le 45^0$. Then we show that a rotation of an arbitrary θ, $0 \le \theta \le 360^0$ can be performed using the algorithms for these special ca ses.

7.1 $\theta = 180^0$

In this case,

$$i' = -i + a$$
$$j' = -j + b$$

The rotation can be performed as follows:

Step 1: [Column reordering] Each processor, p, sets $dest(p) = -i + a$ where i is the row number of the processor. Next, a column reordering as described in Section 2.3.3 is done.

Step 2: [Row reordering] Each processor, p, sets $dest(p) = -j + b$ where j is the column number of the processor. Next, a row reordering as described in Section 2.3.3 is done.

Note that the *dest* values in each column in *Step* 1 and those in each row in *Step* 2 are in decreasing order. *Step* 1 sends all pixels to their correct destination row while *Step* 2 sends them to the correct column. The column and row reordering of *Steps* 1 and 2 can be replaced by column and row reversal followed by a shift. Since a reversal can be done in $O(\log N)$ time, [5], the complexity of a 180^0 rotation on a hypercube is $O(\log N)$ regardless of how the RAW's of *Steps* 1 and 2 are accomplished. On a mesh, its complexity is $O(N)$.

7.2 $\theta = \pm 90^0$

The case $\theta = 90^0$ and $\theta = -90^0$ are quite similar. We consider only the case $\theta = 90^0$. Now,

$i' = -j + a + b$

$j' = i - a + b$

The steps in the rotation are:

Step 1: [Transpose] Transpose the image so that $I^{new}[i, j] = I^{old}[j, i]$

Step 2: [Column Reorder] Each processor sets $dest(p) = a + b - i$ where i is the row number of the processor. Next, a column reordering (cf. Section 2.3.3) is done.

Step 3: [Shift] A rightward shift of $-a + b$ is performed on each row of the image.

Note that in a 90^0 rotation the pixel originally at $[i, j]$ is to be routed to

$[-j + a + b,\ i - a + b]$. *Step* 1 routes the pixel to position $[j,\ i]$; *Step* 2 routes it to $[a + b - j,\ i]$; and *Step* 3 to $[a + b - j,\ i - a + b]$. The transpose of *Step* 1 can be performed on a hypercube in $O(\log N)$ time using the algorithm of [5]. The overall complexity is $O(\log N)$ on a hypercube.. Once again, the column reordering of *Step* 2 can be done by a column reversal followed by a shift. This does not change the asymptotic complexity. On a mesh, the three steps can be completed in O(N) time.

7.3 $|\theta| \le 45^0$

We explicitly consider the case $0 \le \theta \le 45^0$ only. The case $-45^0 \le \theta < 0$ is similar. The steps for the case $0 \le \theta \le 45^0$ are:

Step 1: [Column Reorder] Set $dest(p) = \lceil i\cos\theta - j\sin\theta + A \rceil$ where i is the row number and j the column number of processor p. Since j is the same in a column, $dest(p)$ is non-decreasing in each column. Hence a column reordering can be done as described in Section 2.3.3. All data with the same destination are routed to that destination.

Step 2: [Row Reorder] Set $dest(p) = \lceil i\tan\theta + j sec\theta - A\tan\theta + B \rceil$ where i and j are respectively, the row and column numbers of processor p. A row reordering may be performed as described in Section 2.3.3.

Step 3: [Shift] Pixels that need to be shifted left by one along rows are shifted.

Step 1 sends each pixel to its correct destination row. Since $0 \le \theta \le 45^0$, $1/\sqrt{2} \le \cos\theta \le 1$. Hence, each processor can have at most 2 pixels directed to it. The column reordering of *Step* 1 is done such that both these reach their destination. Following this, the pixel(s) in the proces sor at position $[i,\ j]$ originated in processors in column j and row

$$\frac{i + j\sin\theta - A - \delta}{\cos\theta}$$

where $0 \le \delta < 1$ accounts for the ceiling function in (4). From (4), it follows that these pixels are to be routed to the processors in row i and column $j = \lceil y \rceil$ where y is given by:

$$y = \left(\frac{i + j\sin\theta - A - \delta}{\cos\theta}\right)\sin\theta + j\cos\theta + B$$

$$= i\tan\theta + j\left(\frac{\sin^2\theta + \cos^2\theta}{\cos\theta}\right) - A\tan\theta - \delta\tan\theta + B$$

$$= i\tan\theta + j sec\theta - A\tan\theta + B - \delta\tan\theta$$

In *Step* 2, the pixels are first routed to the column $\lceil i\tan\theta + j sec\theta - A\tan\theta + B \rceil$. Then, in *Step* 3, we account for the $\delta\tan\theta$ term in the formula for y. For $0 \le \theta \le 45^0$, $\tan\theta$ is in the range $[0,\ 1]$. Since $0 \le \delta < 1$, $0 \le \delta\tan\theta < 1$, the pixels need to be shifted leftwards on the rows by at most 1. Note that since $1 \le sec\theta \le \sqrt{2}$ for $0 \le \theta \le 45^0$, $dest(p)$ is different for different processors on the same row. One readily sees that, on a hypercube, $O(\log N)$ time suffices for the rotation, and O(N) time is sufficient on a mesh.

7.4 $0 \le \theta \le 360^0$

Every θ in the range $[0{,}360]$ can be cast into one of the forms:

(a) $-45 \le \theta' \le 45$

(b) $\pm 90 + \theta', \quad -45 \le \theta' \le 45$

(c) $\pm 180 + \theta', \quad -45 \le \theta' \le 45$

Case (a) was handled in the last subsection. Cases (b) and (c) can be done in two steps. First a $\pm 90^0$ or a 180^0 rotation is done (note that a $+180^0$ and a -180^0 rotation are identical). Next a θ' rotation is performed. This two step process may introduce some errors because of end off conditions from the first step. These can be eliminated by implementing all rotations as wraparound rotations and then having a final cleanup step to eliminate the undesired wraparound pixels.

8 Scaling

Scaling an image by s, $s \geq 0$, around position $[a, b]$ requires moving the pixel at position $[i, j]$ to the position $[i', j']$ such that [3]:

$$i' = \lceil si + a(1-s) \rceil$$

$$j' = \lceil si + b(1-s) \rceil$$

$$0 \leq i, j < N.$$

In case $i' \geq N$ or $j' \geq N$, the pixel is discarded. If two or more pixels get mapped to the same location then we have two cases:

(1) only one of these is to survive. The surviving pixel is obtained by some associative operation such as *max*, *min*, *average* etc.

(2) all pixels are to survive.

When $s > 1$, then in addition to routing each pixel to its destination pixel, it is necessary to reconnect the image boundary and fill in the inside of the objects in the image [3]. The pixel routing can be done in $O((\log N)/s)$ time on a hypercube and in $O(N/s)$ time on a mesh when $s < 1$ and all pixels to the same destination are to survive. In all other cases, pixel routing takes $O(\log N)$ time on a hypercube and $O(N)$ time on a mesh. The routing strategy is to perform a row reordering followed by a column reordering. Reconnecting the boundary and filling require $O(\log N)$ time on a hypercube and $O(N)$ time on a mesh.

9 Conclusions

We have developed efficient hypercube and mesh algorithms for image shrinking, expanding, translation, rotation, and scaling. All our algorithms require $O(1)$ memory per PE. The complexity of each operation on an N^2 PE hypercube is $O(d\log N)$ and on an $N \times N$ mesh is $O(dN)$ where d is the maximum number of data that is destined to any processor.

10 Acknowledgement

The research reported in this chapter was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

11 References

[1] T. E. Chan and Y. Saad, "Multigrid algorithms on hypercube multiprocessor", *IEEE Transactions on Computers", Vol. C-35,* **Nov. 86**, pp 969-977.

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

[3] S. Y. Lee, S. Yalamanchali and J. K. Agarwal, "Parallel Image Normalization on a Mesh

Connected Array Processor", *Pattern Recognition*, Vol. 20, No. 1, **Jan 87**, pp. 115-120.

[4] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD computers", *IEEE Transactions on Computers*, No. 2, Vol. C-30, **Feb 1981**, pp. 101-107.

[5] D. Nassimi and S. Sahni, "Optimal BPC permutations on a cube connected computer", *IEEE Transactions on Computers*, No. 4, Vol. C-31, **April 1982**, pp. 338-341.

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

[7] S. Ranka and S. Sahni, "Image Template Matching on an SIMD hypercube multicomputer", *1988 International Conference on Parallel Processing*, Vol III, Algorithms & Applications, Pennsylvania State University Press, pp 84-91.

[8] S. Ranka and S. Sahni, "Image Template Matching on MIMD hypercube multicomputers", *1988 International Conference on Parallel Processing*, Vol III, Algorithms & Applications, Pennsylvania State University Press, pp 92-99.

[9] A. P. Reeves and C. H. Francfort, "Data Mapping and Rotation Functions for the Massively Parallel Processor", *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, **1985**, pp. 412-417.

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

[11] A. Rosenfeld, "A note on shrinking and expanding operations on pyramids", *Pattern Recognition Letters 6*, **Sept 1987**, pp. 241-244.

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