HYPERCUBE ALGORITHMS FOR IMAGE TRANSFORMATIONS [+]

Sanjay Ranka and Sartaj Sahni

University of Minnesota

Abstract

Efficient hypercube 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 and in $O(\log N)$ time on an SIMD hypercube. Translation, rotation, and scaling of an $N \times N$ image take $O(\log N)$ time on an N^2 processor hypercube.

Keywords and Phrases

Hypercube multicomputers, SIMD, MIMD, shrinking, expanding, rotation, translation, scaling.

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

1 INTRODUCTION

Since transformations on two dimensional images are very compute intensive, several authors have developed parallel algorithms for these. For example, Rosenfeld, [ROSE87], 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, [ROSE87] 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 $O(2^k)$. Note that a 2^k step unresampled shrinking/expanding is easily done in $O(2^k)$ time on an $N \times N$ mesh. The unresampled algorithm of [ROSE87] does not generalize to the case of grayscale images. In this paper, 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 and $O(\log N)$ time on an N^2 processor SIMD hypercube.

Lee, Yalamanchali, and Aggarwal, [LEE87], 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.

In the next section we describe our hypercube model and some basic data 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 PRELIMINARIES

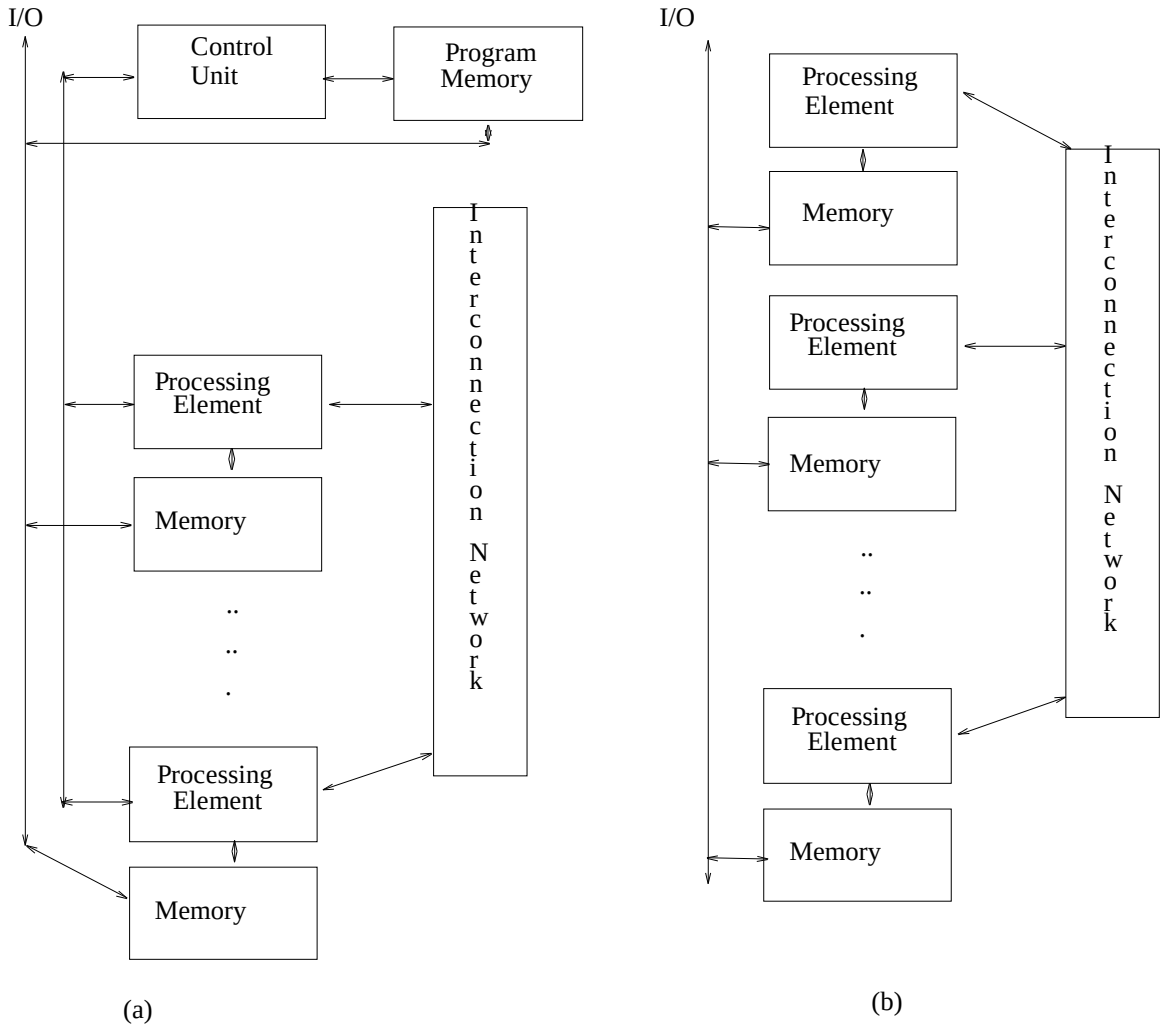
2.1 Hypercube Multicomputer

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 the 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

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



(a) SIMD hypercube
(b) MIMD hypercube
Figure 1: Hypercube multicomputers

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

$(i_0 = 1)$ 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. The topology of a 16 node hypercube interconnection network is shown in Figure 2. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}....i_0$ be the binary

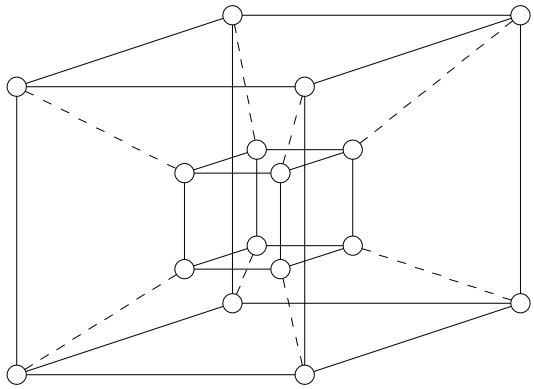


Figure 2 : A 4 dimensional hypercube (16 PEs)

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 $i_{p-1}i_{p-2}...i_0$ is connected to processors $i_{p-1}\cdots\bar{i_k}....i_0$, $0 \le k \le p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

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

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

   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 $i^{(b)}$. Hence the instruction.

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

   takes one unit route, while the instruction:

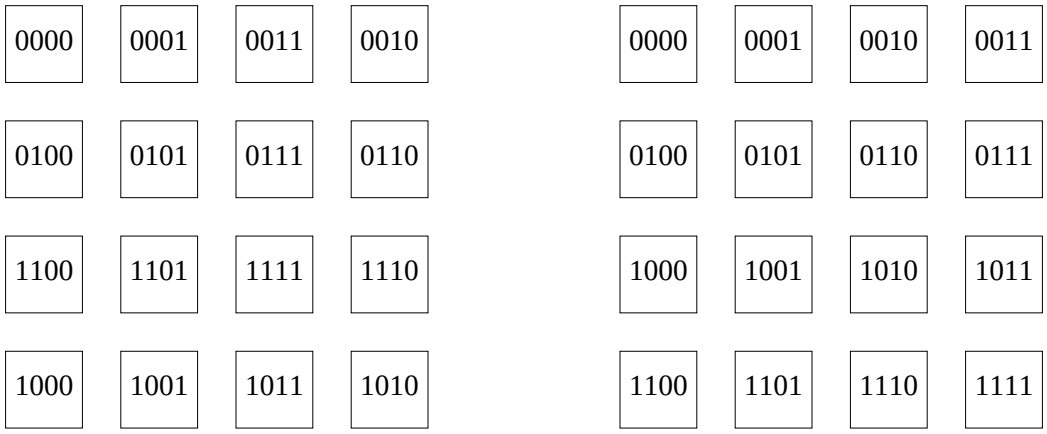
   $B(i^{(2)}) \leftarrow B(i)$

   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 $j^{(a)}$, $a \neq b$.



(a) Gray code mapping (b) Row major mapping

Figure 3 : A 16 PE hypercube viewed as an 4×4 grid

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:

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

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 $S_3 = 000, 001, 011, 010, 110, 111, 101, 100$.

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 [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 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 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 $A(qW + (j-i) \bmod W)$, $0 \le q < (P/W)$. *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 [RANK88a]. 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).

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 [RANK88b] . 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 Row and Column Reordering

These are special cases of the random access write (RAW) operation defined in [NASS81]. We assume an $N \times N$ array logical view of an N^2 PE hypercube (cf. Section 2.2). In a row reordering the destination processor for data in any PE is another PE in the same row. Hence, it is sufficient for each PE to simply have a value *dest*(p) which gives the column index of the destination PE. Furthermore, 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 sort step of the RAW algorithm of [NASS81] may be replaced by a step that does a data concentration. This data concentration takes $O(\log N)$ time [NASS81]. Another change is needed when *dest* is a nonincreasing function. In this case, the ranking step of [NASS81] does a reverse ranking (i.e., right to left rather than left to right). In case *dest*(p) is not in the range $[0, N-1]$, the data from processor p is not routed anywhere. Since the modified RAWs can be done on all rows in parallel, the time required for row reordering is $O(d \log N)$ 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.

Column reordering is the column analog of row reordering. It is performed in an analogous manner.

3 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 [ROSE82] and [ROSE87] to be the $N \times N$ image S^q such that:

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

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

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

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

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

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 [ROSE87], 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 –∞. 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 –∞ 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 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 [ROSE87], we assume $q = 2^k$.

3.1 MIMD algorithm

On an MIMD hypercube R^q for $q = 2^k$ may be computed using the algorithm of Figure 4. The algorithm assumes that the pixel assigned to each processor is in the register I of that processor. The value of R^q corresponding to that pixel position is to be left in the R register of that processor. The computation of R is done in two stages. These are obtained by decomposing (2) into:

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

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

Figure 4 : Computing R^q on an MIMD hypercube

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

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

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, $0 \leq a < k$. 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 an MIMD shift of 1 takes 1 unit route, of 2 takes 2 unit routes, of N/2 takes 4, and remaining power of 2 shifts take 3 routes, it follows that the number of unit routes required by

procedure SHRINK is at most $6k + 2$. Once R^q, $q = 2^k$, has been computed, S^q may be computed using a similar algorithm. This requires an addition $6k + 2$ unit routes. The overall time complexity is $O(k)$.

3.2 SIMD Algorithm

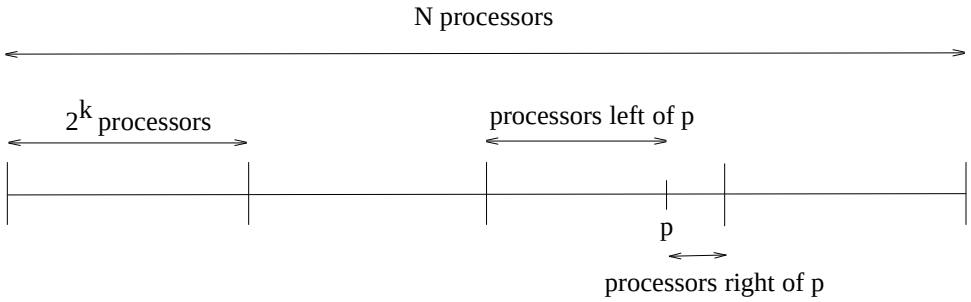


Figure 5 : 2^k blocks of processors

Since a shift of 2^i in a window of size N takes $2\log(N/2^i)$ unit routes, using the algorithm just developed will result in an overall complexity of $O(k\log N)$. 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 Figure 5).

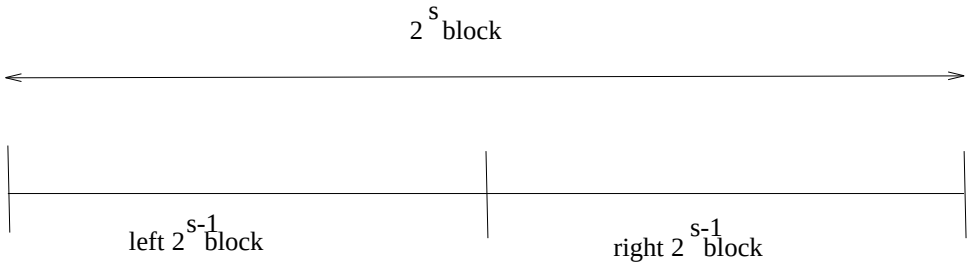


Figure 6 : A 2^s block of processors

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

```
procedure SHRINK;
{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)}
    SHIFT(left, –q, N);
    SHIFT(right, q, N);
    R(p) :=min { R(p), left(p), right(p)}
end;
```

Figure 7 : Computing R^q on an SIMD hypercube

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) = ∞

whole(p) = I(p)

Each 2^s block for $s > 0$ consists of two 2^{s-1} blocks as shown in Figure 6. 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,

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

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

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

The algorithm of Figure 7 implements the strategy just developed. The total number of unit routes for the case $q = 2^k$ is $2k + 2(2\log(N/q)) = 4\log N - 2k$. The total time complexity is $O(\log N)$. As noted earlier, the algorithm is directly applicable to the case when q is not a power of 2. We need simply define $k = \lfloor \log q \rfloor$. The time complexity remains $O(\log N)$.

4 IMAGE 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.

5 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 [REEV85]:

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

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

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.

5.1 $\theta = 180^0$

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, [NASS82], the complexity of a 180^0 rotation is $O(\log N)$ regardless of how the RAW's of *Steps* 1 and 2 are accomplished.

5.2 $\theta = \pm 90^0$

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 in $O(\log N)$ time using the algorithm of [NASS82]. The overall complexity is $O(\log N)$. 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.

5.3 $|\theta| \le 45^0$

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:

$$\begin{aligned} y &= \left(\frac{i + j\sin\theta - A - \delta}{\cos\theta}\right)\sin\theta + j\cos\theta + B \\ &= i\tan\theta + j\left(\frac{\sin^2\theta + \cos^2\theta}{\cos\theta}\right) - A\tan\theta - \delta\tan\theta + B \\ &= i\tan\theta + j\sec\theta - A\tan\theta + B - \delta\tan\theta \end{aligned}$$

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 $O(\log N)$ time suffices for the rotation.

5.4 $0 \le \theta \le 360^0$

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.

6 SCALING

Scaling an image by s, $s \ge 0$, around position $[a, b]$ requires moving the pixel at position $[i, j]$ to the position $[i', j']$ such that [LEE87]:

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

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

$0 \le i, j < N$.

In case $i' \ge N$ or $j' \ge N$, 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 [LEE87]. The pixel routing can be done in $O((\log N)/s)$ time when $s < 1$ and all pixels to the same destination are to

survive. In all other cases, pixel routing takes $O(\log N)$ time. The routing strategy is to perform a row reordering followed by a column reordering. Reconnecting the boundary and filling require only $O(\log N)$ time.

7 CONCLUSIONS

We have developed efficient hypercube algorithms for image shrinking, expanding, translation, rotation, and scaling. For the case of shrinking and expanding, we have developed different algorithms for SIMD and MIMD hypercubes. For the remaining operations our algorithms are the same for both SIMD and MIMD hypercubes. All our algorithms require $O(1)$ memory per PE. The algorithms for translation, rotation and scaling are readily applied to mesh connected computers also. The complexity on an $N \times N$ mesh is $O(dN)$ where d is the maximum number of data that is destined to any processor. Generally, $d = 1$ and the complexity becomes $O(N)$.

8 REFERENCES

[CHAN86] T. E. Chan and Y. Saad, "Multigrid algorithms on hypercube multiprocessor", *IEEE Transactions on Computers", Vol. C-35,* **Nov. 86**, pp 969-977.

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

[LEE87] 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.

[NASS81] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD computers", *IEEE Transactions on Computers*, No. 2, Vol. C-30, **Feb 1981**, pp. 101-107.

[NASS82] 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.

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

[RANK88a] 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.

[RANK88b] 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.

[REEV85] 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.

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

[ROSE87] A. Rosenfeld, "A note on shrinking and expanding operations on pyramids", *Pattern Recognition Letters 6,* **Sept 1987**, pp. 241-244.

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