n^3 프로세서 재구성가능 메쉬에서 n^2 화소 이진영상과 경계코드간의 효율적인 변환

김 명[†]

요 약

본 논문에서는 $n \times n \times n$ 프로세서로 구성된 재구성가능 메쉬에서 $n \times n$ 개의 화소가 있는 이진영상을 경계코드로 변환하거나 그 역변환을 하는 알고리즘을 제안한다. 이와 동일한 변환을 하는 $O(1)$ 시간 알고리즘들이 이미 제안되었는데, 이들이 사용하는 프로세서의 수는 $O(n^4)$으로, 영상의 화소 수와 비교해 볼 때 지나치게 많다고 하겠다. 본 논문에서는 n^3 개의 프로세서만을 사용하는 속도 빠른 변환 알고리즘을 소개한다. 여기서 제안하는 경계코드를 이진영상으로 변환하는 알고리즘의 실행시간은 $O(1)$이고, 그 역변환 알고리즘의 실행시간은 $O(\log n)$이다.

Efficient Transformations Between an n^2 Pixel Binary Image and a Boundary Code on an n^3 Processor Reconfigurable Mesh

Myung Kim[†]

ABSTRACT

In this paper, we present efficient reconfigurable mesh algorithms for transforming between a binary image and its corresponding boundary code. These algorithms use $n \times n \times n$ processors when the size of the binary image is $n \times n$. Recent published results show that these transformations can be done in $O(1)$ time using $O(n^4)$ processors. The number of processors used by these algorithms is very large compared to the number of pixels in the image. Here, we present fast transformation algorithms which use n^3 processors only. The transformation from a boundary code to a binary image takes $O(1)$ time, and the converse transformation takes $O(\log n)$ time.

1. Introduction

Image representation is an important field of research in computer vision, image processing, computer graphics, geographic information systems, and cartography applications. For different applications, different image representations have been proposed. These include boundary codes, quadtrees, and run length codes [14]. Since each representation has advantages and disadvantages, it is of great interest to develop efficient algorithms for transforming between any pair of such representations.

Recently, parallel algorithms for such transformations were developed on architectures such

※ This research was supported by Ewha Research Grants.
† 정 회 원 : 이화여자대학교 컴퓨터학과 교수
논문접수 : 1998년 6월 3일, 심사완료 : 1998년 6월 29일

as mesh [3], hypercube [1, 2, 4, 8], and reconfigurable mesh [7, 9]. In this paper, we focus on the transformation between a binary image and a boundary code, and present efficient algorithms on a reconfigurable mesh of size $n \times n \times n$ when the number of pixels in the binary image is $n \times n$.

Let us first define some terms that will be used throughout the paper [14]. Consider a binary image of $n \times n$ pixels. The pixel in row i and column j is represented either by its row and column numbers as $P(i, j)$ or by its row major number as $P(i \times n + j)$. Two pixels in a binary image are said to be *4-adjacent* if they share a side. Two pixels are said to be *8-adjacent* if they share a side or a corner. For example, in Fig. 1. $P(46)$ and $P(56)$ are 4-adjacent. $P(46)$ and $P(55)$ are 8-adjacent but not 4-adjacent. A set of pixels, S, is said to be *4-connected (8-connected)*, if for any pixel p and q in S, there exists a sequence of pixels, $p = p_0, p_1, p_2, \ldots, p_k = q$ in S, such that p_{i+1} is 4-adjacent (8-adjacent) to p_i, $0 \leq i < k$.

A *(black) region* is defined as a maximal set of 8-connected black pixels. A *(white) hole* is a maximal set of 4-connected white pixels which are surrounded by a black region. A black pixel in a region is said to be a *bound ary pixel* of t he region, if it shares a side with either a white pixel or the image frame. The *boundary of a region* is the set of such sides of its boundary pixels. The boundary of a black region forms one or more simple paths, and we call each such path as a *cycle*. For example, in Fig. 1, $P(22)$, $P(31)$, $P(33)$, $P(42)$ form a black region. $P(32)$ forms a hole. The boundary of this black regio n is described by two cycles.

A *(4-directional) boundary code* of a binary image, is a sequence, $C_1C_2 \ldots C_t$, where t is the number of cycles in the image, and C_i is a sequence of characters describing cycle i. Each C_i is called a *cycle code*. A cycle code begins with "*". It is followed by the row and column numbers of a pixel whose upper left corner is the *start point* of the cycle. A pixel whose upper left corner is the start point of a cycle is called the *start pixel* of the cycle. These coordinates are followed by a sequence of unit vectors (links): R, D, L, and U, each representing the direction: right, down, left, or up. The first unit vector in a cycle code is called as the *start link*.

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	10	11	12	13	14		16		18	19
2	20	21		23	24	25		27		29
3	30		32		34	35	36	37	38	39
4	40	41		43	44	45			48	49
5	50	51	52	53	54					59
6	60	61		63		65				69
7	70		72		74		76	77		79
8	80	81		83	84	85			88	89
9	90	91	99	93	94	95	96	97	98	99

(Fig. 1) A 10×10 binary image.

If a cycle is on the outside boundary of a region, its cycle code describes the boundary in a clockwise motion from the start point; otherwise, it is in a counterclockwise motion. Thus, in either case, the boundary pixels are on the right side of the sequence in the principal direction. A boundary code for the image in Fig. 1 can be given as follows:

* 2 2 R D R D L D L U L U R U * 1 5 R D R
U R D R D L U L D L U L U * 4 3 U L D R
* 4 6 R R D R D D D L D L L U L U L D L
D L U L U R U R D R U R U R U * 7 2 D R
U L * 6 5 D R U L * 7 7 L D R R U L

There are 7 cycles in this boundary code. The cycles begin at $P(1, 5)$, $P(2, 2)$ and $P(4, 6)$ describe the outside boundaries of the black regions. The remaining 4 cycles describe the holes in the image. For example, the first cycle in the boundary code begins at the upper left corner of $P(2, 2)$. This cycle begins with an R link followed by a D link followed by an R link, and so on.

Recently, researchers show much attention to designing parallel algorithms for transforming between any pair of the following binary image representations: binary images, boundary codes, run length codes, and quadtrees [1, 2, 3, 4, 7, 8, 9]. For the transformation between a binary image and a quadtree, Hung, Rosenfeld [3] presented mesh algorithms, Dehni, Ferreira, Rau-Chaplin [1] and Ibarra, Kim [4] gave SIMD hypercube algorithms, and Kim, Jang[9] gave reconfigurable mesh algorithms.

For the transformation between a boundary code and a quadtree, various SIMD hypercube algorithms were developed. Dehni, Ferreira, Rau-Chaplin [1] gave an algorithm for building a quadtree from a boundary code. Doctor and Sudborough [2] gave a randomized algorithm for the same transformation. Both algorithms do not allow holes inside the black regions. Kim, Ibarra [8] presented algorithms for transforming between a boundary code and a quadtree. These run faster than the previous algorithms [1, 2] and allow holes inside the black regions. In [8], Kim, Ibarra also gave algorithms for transforming between any pair of binary images, boundary codes, quadtrees, and run length codes.

Kim [7] recently presented $O(1)$ time algorithms for transforming between a binary image and a boundary code on a reconfigurable mesh. The transformation from a boundary code of length b to its corresponding binary image of size $n \times n$ uses $n \times n \times \max(n, b)$ processors, and its converse algorithm uses $2n^2 \times n \times 2n$ processors. Although these algorithms run in constant time, the number of required processors does not seem to be small enough to be used in practice.

In this paper, we present reconfigurable mesh algorithms for the same transformation. Our main objective is to reduce the number of processors by a factor of $O(n)$ while keeping the execution time reasonably small. We use $n \times n \times n$ processors, as for the transformations between a binary image and a quadtree in [9]. Our algorithm for transforming a boundary code to its corresponding binary image takes $O(1)$ time and its converse algorithm takes $O(\log n)$ time.

The paper is organized as follows. In section 2, we give a brief explanation on the computing model used for the transformation algorithms. Some constant time operations are also mentioned in this section. In section 3, we present an algorithm for transforming a binary image to a boundary code. Its converse algorithm is given in section 4. Section 5 concludes the paper.

2. Preliminaries

The computing model used for our algorithms is a reconfigurable mesh [10]. A 2-D $n \times n$ reconfigurable mesh consists of $n \times n$ processors (PEs) which are connected to a grid-shaped reconfigurable broadcast bus. Each PE has four I/O ports: N, S, E, W. The internal connections among the 4 ports in a PE can be configured during the execution of an algorithm. This allows the broadcast bus to be divided into sub-

buses, providing smaller reconfigurable meshes. It is assumed that a value broadcast consists of $O(\log n)$ bits and takes $O(1)$ time, as is the assumption in [15]. It is not allowed for more than one PEs to broadcast to a subbus shared by multiple PEs at any given time. Each PE has $O(1)$ words and can perform an arithmetic (or a logic) operation in $O(1)$ time.

Internal port connections of a PE in a 2-D reconfigurable mesh can be realized in 15 different ways [6, 7, 11]. Assume that the notation, {NS, EW}, represents the situation that the N port is connected to the S port and the E port is connected to the W port. The 15 possible port connections can be described as follows: {N, S, E, W}, {NS, E, W}, {N, S, EW}, {NE, S, W}, {N, ES, W}, {N, E, SW}, {NW, E, S}, {NE, SW}, {NW, ES}, {NS, EW}, {NEW, S}, {NES, W}, {N, ESW}, {NSW, E}, {NESW}.

A 3-D reconfigurable mesh is defined similarly. Each PE in the mesh of size $n\times n\times n$ is indexed as a triple: (l, r, c), where $0\le l, r, c < n$. Here, l represents the layer on which the corresponding PE is located, r and c are the row and column numbers of the PE on layer l. Each PE has two I/O ports along each dimension (a total of 6 I/O ports per PE). The U and D ports are along dimension 0, the E and W ports are along dimension 1, and the N and S ports are along dimension 2. The buses formed along dimensions 0, 1, and 2 are called the UD bus (layer bus), the EW bus (row bus), and the NS bus (column bus), respectively.

Let us now describe some reconfigurable mesh operations that will be used in our algorithms.

Observation 1.
On an $n\times n$ reconfigurable mesh, n numbers stored in a row (or a column) can be permuted into any order in $O(1)$ time [9].

Observation 2.
An $n\times n\times n$ 3-D reconfigurable mesh can be seen as an $n\times n^2$ 2-D reconfigurable mesh. This is because there is a dilation-1 embedding from an $n\times n^2$ 2-D mesh into an $n\times n\times n$ 3-D mesh. Thus, it is possible to run $n\times n^2$ (or $n^2\times n$) 2-D reconfigurable mesh algorithms on an $n\times n\times n$ 3-D reconfigurable mesh without increasing their time complexities [9].

Definition 1.
Given n numbers, $a_0, a_1, a_2, \ldots, a_{n-1}$, the prefix sums operation is to compute n quantities, $\sum_{i=0}^{k} a_i$, for all $0\le k < n$.

Observation 3.
On a $\sqrt{n}\times n$ reconfigurable mesh, the prefix sums of an n element binary sequence can be computed in $O(1)$ time [10, 13].

Observation 4.
The prefix sums of a sequence of n integers in the range of 0 to n^2 can be computed in $O(1)$ time on an $n^2\times n$ reconfigurable mesh [13].

Observation 5.
The prefix sums of a sequence of n integers in the range of 0 to n^c can be computed in $O(c)$ time on an $n\times n$ reconfigurable mesh [13].

Observation 6.
For $0\le p\le n^2$, p numbers distributed over $n\times n$ PEs on layer 0 of a reconfigurable mesh of size $n\times n\times n$ can be collected, in $O(1)$ time, into the first p PEs of layer 0 retaining the original order among the numbers [9].

Observation 7.
On an $n \times n$ reconfigurable mesh, n numbers can be sorted in $O(1)$ time [6, 10].

Observation 8.
On an $n \times n \times n$ reconfigurable mesh, n^2 numbers can be sorted in $O(1)$ time [5].

2. From binary image to boundary code

The reconfigurable mesh used for our algorithm consists of $n \times n \times n$ PEs. Each PE in the mesh is indexed by a triple (l, r, c) as defined in section 2. The input binary image of size $n \times n$ is initially stored on the bottom layer, which is layer 0. Pixel $P(i, j)$ is stored in PE(0, i, j), $0 \leq i, j < n$. For simplicity, it is assumed that the pixels in row $n-1$ or column $n-1$ are white. From this input configuration, we generate a boundary code, and put it in linear order with the first link at PE(0, 0, 0). That is, we store the k-th link, $0 \leq k < 2n^2$, of the boundary code to PE(l, i, j), where $l = k$ div n^2, $i = (k - (l \times n^2))$ div n, $j = k - (l \times n^2 + i \times n)$. The first symbol of a cycle, "*", and the following row, column numbers are stored in the PEs with the corresponding cycle start link.

The algorithm can be briefly described as follows:

Algorithm BinImageToBcode

Phase 1. Each PE(0, i, j), $0 \leq i, j < n$, generates the links that are either on the top side or on the left side of pixel $P(i, j)$. An ID number is then assigned to each link.

Phase 2. Each PE determines the ID of the predecessor link for each of its links.

Phase 3. Each PE determines, for each of its links, the ID of the cycle start link and the distance from the start link to its link.

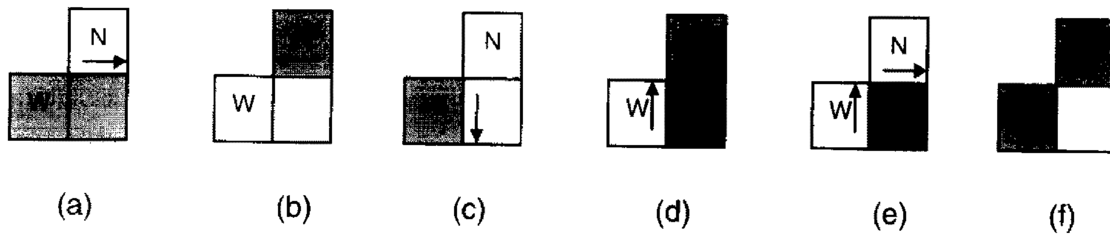
Phase 4. Compute the length of each cycle.

Phase 5. Determine the destination PE address for each start link.

Phase 6. Determine the destination PE addresses for the remaining links. Move all the links to their destination PEs.

Let us explain the algorithm in detail. Phase 1 generates all the links that belong to the output boundary code. The links that are either on the top side or on the left side of $P(i, j)$ are generated by PE(0, i, j), $0 \leq i, j < n$. In order to generate the links, each PE needs to know the colors of its N, W neighbor pixels. Each PE(0, i, j), $1 \leq i, j < n$, thus fetches the colors of the pixels in PE(0, i-1, j) and PE(0, i, j-1). The PEs in row 0 (or column 0) assume that their W neighbor (or N neighbor) is white. An R link is generated by PE(0, i, j) if its pixel is black and its N neighbor is white. It is because a cycle code describes the boundary of a black region in a clockwise motion from the start point. Similarly, a U link is generated if its pixel is black and its W neighbor is white. An L link is generated if its pixel is white and its N neighbor is black. A D link is generated if its pixel is white and its W neighbor is black. Fig. 2 shows the 6 possible cases of link generation in PE(0, i, j). In the figure, the pixel at the lower right corner is $P(i, j)$. Note that the number of links generated by each PE is at most 2.

We next assign an ID number to each link. The ID of a link is a tuple (*Pid*, *LinkType*).

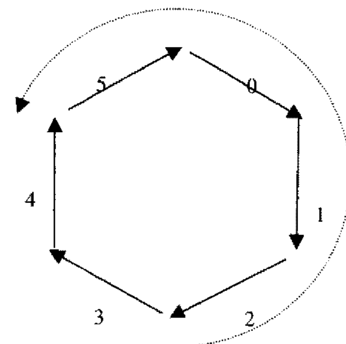


(Fig. 2) Link generation in PE(0, i, j).

where *Pid* is the linear number of the PE where the link has been generated, and *LinkType* is 0 if it is an R or an L, 1 otherwise. A link can be identified by its ID. If a link is of *LinkType* x, $x=0, 1$, we call it *type x link*. For example, the ID of an R link generated in PE(0, 2, 3) is ($0\times n^2+2\times n+3$, 0).

At this moment, although we know which links belong to the output boundary code, we do not know how many cycles there are in the boundary code, which cycle each link belongs to, and in what order the links should be arranged in the output boundary code. As the first step to solve these, we determine, in phase 2, the ID of the predecessor link of each generated link.

Let us define some more terms. Suppose that $c_0c_1c_2...c_{k-1}$ is a cycle code, where c_i is the i-th link in the cycle. The *length* of a cycle is defined to be the number of links in the cycle. In this example, the cycle length is k. The *predecessor link* of c_i is defined to be $c_{(i-1+k) \bmod k}$. That is, the predecessor of c_i, $1\le i< k$, is c_{i-1} and the predecessor of c_0 is c_{k-1}. The x *preceding links* of c_i are all the links (with repetition) we meet when we walk x steps on the region boundary in a counterclockwise motion (in reverse order), starting at $c_{(i-1+k) \bmod k}$. Here, x is allowed to be larger than k. For example, consider the cycle in Fig. 3. The cycle is represented as a ring. Its cycle code is $c_0c_1c_2c_3c_4c_5$. The 4 preceding links of c_3 are c_2, c_1, c_0, c_5. The 8 preceding links of c_3 are c_2, c_1, c_0, c_5, c_4, c_3, c_2, c_1. For two links c_i and c_j, $0\le i,j< k$, the *distance* from c_i to c_j, is defined to be the number of links we meet when we walk on the region boundary in a clockwise motion, starting at $c_{(i+1) \bmod k}$ and ending at c_j. That is, the *distance* from c_i to c_j is $j-i$ if $i\le j$, $j-i+k$ otherwise. In Fig. 3, the distance from c_5 to c_3 is 4 and the distance from c_3 to c_5 is 2.

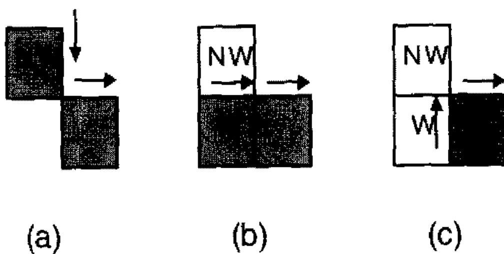


(Fig. 3) The 4 preceding links of c_3.

In order to determine the predecessor link of a link, we need to know the colors of its neighbor pixels. Thus, each PE(0, i, j), $0\le i,j< n$, first fetches the colors of the eight neighbor pixels of $P(i,j)$, and uses the following rules to locate the predecessor link of each of its links. Suppose that PE(0, i, j) has

an R link. It tells us that its pixel is black and its N neighbor is white. The predecessor link is a D link in PE(0, i-1, j) if the NW neighbor is also black, as in Fig. 4(a). If the NW neighbor is white, PE(0, i, j) checks the color of its W neighbor. If it is black, the predecessor link is an R link in PE(0, i, j-1), as in Fig. 4(b). If both NW and W neighbors are white, then the predecessor link is a U link in PE(0, i, j), as in Fig. 4(c). From the calculated information, the ID of the predecessor link can be easily obtained. The ID of the predecessor links of L, D, U links can be determined similarly.

In phase 3, each PE determines, for each of its links LK, the ID of the cycle start link, and computes the distance from the cycle start link to LK. The cycle start link can be any link in the cycle. So, in our algorithms, we assume that the start link is the link with smaller ID than any links in the cycle. Note that $ID_1 = (Pid_1, LinkType_1)$ is said to be *smaller* than $ID_2 = (Pid_2, LinkType_2)$ if either ($Pid_1 < Pid_2$) or ($Pid_1 = Pid_2$ and $LinkType_1 < LinkType_2$).



(Fig. 4) The predecessor link of an R link.

Phase 3 uses a technique called "pointer jumping". The pointer jumping process is divided into $2\log n + 2$ steps. In step d, $0 \le d \le 2\log n + 1$, each PE(0, i, j) chooses, for each of its links LK, a candidate cycle start link from the 2^d preceding links of LK. PE(0, i, j) also computes the distance from the chosen link to LK. The candidate link is the link with smaller ID than any links among those 2^d links. After $2\log n + 2$ such steps, most recently chosen candidate cycle start links become the correct start links of the corresponding cycles, since a cycle is not longer than $2n^2$.

Here are the details of step d, $0 \le d \le 2\log n + 1$. Each PE(0, i, j) keeps 3 data items for each of its links LK : $Start(LK)$, $Dist(LK)$, and $Pred(LK)$. Let us first explain what has been calculated in step d-1: (1) $Start(LK)$ is the ID of the candidate cycle start link chosen from the 2^{d-1} preceding links of LK. (2) $Dist(LK)$ is the distance from $Start(LK)$ to LK. (3) $Pred(LK)$ is the ID of the link which we meet at the 2^{d-1}-th step when we walk on the region boundary in a counterclockwise motion, starting at the predecessor link of LK. In step d, each PE(0, i, j) updates the 3 data items as follows:

(1) $Start(LK) = \min\{Start(LK), Start(Pred(LK))\}$

(2) $Dist(LK) = Dist(LK)$, if $Start(LK)$ has not been updated

$= Dist(Pred(LK)) + 2^{d-1}$, otherwise.

(3) $Pred(LK) = Pred(Pred(LK))$

As we can see here, in order to carry out the above calculations, each PE(0, i, j) needs to obtain $Start(Pred(LK))$, $Dist(Pred(LK))$, $Pred(Pred(LK))$ from the PE which has generated $Pred(LK)$. Let us now focus on such data communications. Currently, there are as many as n^2 PEs which need to get data from other PEs. Some of them may have two links to process. And, some PEs may receive more than one data

requests from other PEs. Since we plan to use the constant time sorting operation (observation 8), we organize the data requests so that at any given time PEs send (or receive) at most one data request. This process is divided into 4 substeps. In substep s, $1 \leq s \leq 4$, only the PEs, PE(0, i, j), that satisfy condition s send out their requests, where the conditions are given below. Assume that *LK* is a link in PE(0, i, j).

condition 1: *LK* is of type 0 and *Pred*(*LK*) is of type 0.
condition 2: *LK* is of type 0 and *Pred*(*LK*) is of type 1.
condition 3: *LK* is of type 1 and *Pred*(*LK*) is of type 0.
condition 4: *LK* is of type 1 and *Pred*(*LK*) is of type 1.

Next, we show how substep 1 can be done in $O(1)$ time. Each PE(0, i, j) sends out a data request for its link *LK*, if it is of type 0 and *Pred*(*LK*) is also of type 0. PE(0, i, j) first prepares a record R = {*DestPid*, *Info*}, where *DestPid* is the address of the PE which generated *Pred*(*LK*). The PEs which do not satisfy condition 1 set their *DestPid* to n^2. Note that if PE(0, i, j) has a link of type 0, the value of its *DestPid* is less than n^2. *Info* contains the remaining information for the data request. Note that our objective is to move R to PE(0, *DestPid* div n, *DestPid* mod n). After the records are prepared, they are sorted in increasing order of their *DestPid* using $n \times n \times n$ PEs (observation 8).

The sorted R's are stored on the bottom layer, one R per PE. The records are next moved straight up to the r-th layer of the mesh, where r = *DestPid* div n. Since the number of records on each layer is not more than n, they do not take more than two adjacent rows inside the layer. The R's on layer r are then moved to PE(r, r, c), where c = *DestPid* mod n. These are next moved straight down to PE(0, r, c) using the layer bus of the mesh. This completes the data request of substep 1. Substeps 2-4 are done similarly. After the data request for link *LK* is sent to the PE with *Pred*(*LK*), we use similar operations to bring the requested data back to the PE with *LK*.

In phase 4, each PE with a start link computes the length of its cycle. This value is later used to allocate the PEs for storing the corresponding cycle code. Note that the length of a cycle is the distance from the start link to the last link of the cycle, which in fact is the predecessor of the start link. Since the predecessor of a link is stored in one of its 8 neighbor PEs, each PE with a start link can easily obtain the cycle length in constant time.

Phase 5 is to determine the destination PE address for each start link. Let $C_{i,j}$ be a cycle whose start link is in PE(0, i, j). Our boundary code will be placed in such a way that $C_{i,j}$ appears before $C_{k,m}$ if $i \times n + j$ is smaller than $k \times n + m$. Note that because of the way our cycle start link is defined, there is at most one start link in each PE. They are either an R link or an L link. In order to determine the PE address for each start link, we use the prefix sum operation. PE(0, i, j) uses variable $L_{i,j}$ to store its cycle length. If PE(0, i, j) does not have a start link, $L_{i,j}$ is 0.

Phase 5 is divided into 5 steps. In step 1, we compute the prefix sums of $L_{i,0}$, $L_{i,1}$, ... , $L_{i,n-1}$, for each row i, $0 \leq i < n$. For computing the prefix sums of the n numbers in row i, we use $n \times n$ PE(l, i, j)'s, for all

$0 \le l < n$ and $0 \le j < n$. The computed $\sum_{j=0}^{k} L_{i,j}$ is stored in $PSUM_{i,k}$ in PE(0, i, k), for each i and k, $0 \le i, k < n$. By observation 5, it takes constant time to compute the prefix sums of n integers in the range of 0 to $2n^2$ using $n \times n$ PEs.

In step 2, $PSUM_{i,n-1}$, for all $0 \le i < n$, is copied to the first column inside the corresponding row. That is, $PSUM_{i,n-1}$ is copied to variable $ROWSUM_i$ in PE(0, i, 0), for all $0 \le i < n$. Note that the value of $ROWSUM_i$, for all $0 \le i < n$, is also in the range of 0 to $2n^2$. And there are n such numbers. Now, the prefix sums of the n $ROWSUM_i$, for all $0 \le i < n$, are computed and stored in SUM_i in PE(0, i, 0), for all $0 \le i < n$.

In step 3, each PE(0, i, 0) on the bottom layer sends the value of SUM_i to all the PEs inside the row through its row bus. Finally, each PE with a start link computes $SUM_i - ROWSUM_i + PSUM_{i,j} - L_{i,j}$ which is the destination PE address of the start link.

Phase 6 is divided into 3 steps. The first step begins with moving the links to their destination layer. If there are more than n^2 links in the output boundary code, the links except the first n^2 links of the boundary code are placed on layer 1. So, we first locate the link whose destination PE is PE($0, n-1, n-1$). This can be done by the PEs with a start link, since they know their link's destination PE address and the cycle length.

Suppose the link we located is the k-th link of cycle $C_{i,j}$. Then PE(0, i, j) prepares a record R = (*StartPid*, *Distance*). *StartPid* is set to $i \times n + j$, which is the index of the PE with the cycle start link. *Distance* is set to k. Now, we set up a bus so that all the PEs on layer 0 are connected in a snake-like fashion. PE(0, i, j) then broadcasts the prepared record R through the established bus. After receiving R, all the PEs compare *StartPid* with its *Start*(LK) and *Distance* with its *Dist*(LK), for each of its link LK. Link LK should be moved to layer 1 if either *StartPid* < *Start*(LK) or (*StartPid* = *Start*(LK) and *Distance* < *Dist*(LK)). Now, using the layer bus, all the links whose destination is on layer 1 are moved straight up to layer 1.

The objective of step 2 is to move all the links on layer 0 to their destination PEs. Here we only explain how to move the links on layer 0. The links on layer 1 are handled similarly. Each PE(0, i, j) first prepares a record R = (*Start*(LK), *Dest*(LK), *Dist*(LK), LK) for each of its links LK. *Start*(LK) is the ID of the start link of the cycle to which LK belongs. *Dest*(LK) is the destination PE address of LK if LK is the start link, undefined otherwise. *Dist*(LK) is the distance from the cycle start link to LK.

At this moment, some PEs may have more than one links (i.e., more than one records). Using the fact in observation 6, we first move the records for type 0 links to low numbered PEs inside the layer, one record per PE. Next, the records for type 1 links are moved to high numbered PEs inside the layer. Since there are at most n^2 records, there is no PE which has more than one records.

In step 3, we sort all the records on layer 0 in increasing order of their *Start*(LK). The records in odd numbered rows are then reversed. Now we set up the bus so that all the PEs on layer 0 are connected in a snake-like fashion. Note that all the records for the links that belong to the same cycle

make a run. What we try to do next is to broadcast the destination PE address of the cycle start link, *Dest(Start(LK))* to all the PEs that have records for the same cycle. In order to do so, each PE first checks its right/left neighbor PEs and disconnects the bus if they belong to different cycles. Now, the PEs with a start link send out the destination PE address of its link through the bus. Each PE then calculates the destination PE address of its link, *Dest(LK)*, by adding the received address to its *Dist(LK)*. All the records are sorted one more time in increasing order of their *Dest(LK)*. The records in the sorted sequence can easily be placed to their destination PE.

As the final step of the algorithm, the row, column numbers of each start link should be generated. Each PE which currently has the start link of a cycle computes these numbers from *Pid* of its *LK*. If the start link is an R link, the computed row, column numbers are correct. If it is an L link, the column number should be increased by 1 since the start point of a cycle is the upper left corner of the pixel from which the cycle begins.

Consider the time taken by each phase of the algorithm. It is obvious that phases 1-2 and 4-6 take $O(1)$ time. Phase 3 consists of $O(\log n)$ steps of pointer jumping. Since each step of phase 3 takes $O(1)$ time, the total time taken by the algorithm is $O(\log n)$. Thus, we can conclude the following:

Theorem 1. *The above algorithm transforms an $n \times n$ binary image to a boundary code. It runs in $O(\log n)$ time using a reconfigurable mesh of size $n \times n \times n$.*

4. From boundary code to binary image

Here, we present an algorithm for transforming a boundary code to its corresponding binary image. The reconfigurable mesh used for the algorithm consists of $n \times n \times n$ PEs. The input boundary code is stored the same way as the output boundary code of the converse algorithm in section 3. From this, we build a binary image and store pixel $P(i, j)$ to PE(0, *i*, *j*), $0 \leq i, j < n$.

The algorithm can be briefly described as follows:

Algorithm BcodeToBinImage

Phase 1. Each PE with a link determines the numbers of U, D, R, L links in between its link and the start link of its cycle.

Phase 2. Each PE with a U link or a D link generates the black boundary pixel that is adjacent to its link.

Phase 3. Move the boundary pixels to their destination PEs.

Phase 4. Generate the remaining black pixels.

The objective of each phase of the algorithm is similar to that in [7]. However, we implement the algorithm using less number of PEs. Note that the algorithm in [7] uses $2n^4$ PEs in the worst case. The details of our algorithm can be described as follows.

In phase 1, each PE(*l*, *i*, *j*), $0 \leq l \leq 1$, $0 \leq i, j < n$, determines the numbers of U, D, R, L links in between its link and the start link of its cycle. Let us first explain how to count the number of U links. The prefix sum operation can be used for the counting. Each PE(*l*, *i*, *j*), $0 \leq l \leq 1$, $0 \leq i, j < n$, sets a flag, $F_{l,i,j}$ to 1 if it has a U link, 0 otherwise. The flags on the bottom two

layers form a binary sequence of length $2n^2$. We first compute the prefix sums of the flags on the bottom layer, and store the computed prefix sum, $\sum_{u=0}^{i} \sum_{v=0}^{j} F_{0,u,v}$, to variable $USUM_{0,i,j}$ in PE(0, i, j), $0 \le i,j < n$. Note that the prefix sum computation of a binary sequence of length n^2 can be done in $O(1)$ time on a reconfigurable mesh of size $n \times n^2$ (observation 3).

We next compute the prefix sums of the flags on layer 1 and store the prefix sums, $\sum_{u=0}^{i} \sum_{v=0}^{j} F_{1,u,v}$, to variable $USUM_{1,i,j}$ in PE(1, i, j), $0 \le i,j < n$. $USUM_{0,n-1,n-1}$, which is stored in PE(0, n-1, n-1), is then broadcasted to all the PEs on layer 1. This value is added to each $USUM_{1,i,j}$, $0 \le i,j < n$, to get the actual prefix sum values. The numbers of D, L, and R links in between the start link and each link of the cycle are computed similarly. The computed prefix sums are then stored in variables, $DSUM_{l,i,j}$, $LSUM_{l,i,j}$, $RSUM_{l,i,j}$, in PE(l, i, j), $0 \le l \le 1$, $0 \le i,j < n$, respectively.

Phase 2 is divided into 3 steps. Step 1 is to rearrange the links so that two adjacent links in the boundary code are stored in two adjacent PEs when the PEs are connected in a snake-like fashion. For simplicity, assume that n is even. Assume also that $LK(l, i, j)$ represents the link which was initially in PE(l, i, j). We rearrange the links of odd numbered rows in reverse order. That is, $LK(l,i,j)$, $0 \le l \le 1$, $0 \le i,j < n$, is moved to PE(l, i, n-j-1). Next, the rows on layer 1 are reversed. That is, for all $0 \le i < n$, $LK(1,i,j)$ is moved to PE(1, n-i-1, j).

In step 2, we set up a subbus that goes through all the PEs with a link that belongs to the same cycle. It can be easily done since two adjacent links in the boundary code are already stored in two adjacent PEs. After the bus connections are set up, the row and column numbers of the start link of each cycle are broadcasted through the subbus. Each PE(l, i, j), $0 \le l \le 1$, $0 \le i,j < n$, uses the received numbers along with the values in $USUM_{l,i,j}$, $DSUM_{l,i,j}$, $LSUM_{l,i,j}$, $RSUM_{l,i,j}$ to determine the row, column numbers of the boundary pixel which is adjacent to its link.

In step 3, we eliminate the black boundary pixels that were generated from an L or an R link. These pixels are not used for generating the remaining pixels. Before moving the boundary pixels to their destination PEs, we assign a type to the remaining boundary pixels: the type is U if it is generated from a U link; the type is D if it is generated from a D link.

The typed boundary pixels are moved to their destination PEs in phase 3. Phase 3 is divided into 4 steps. In step 1, each PE(l, i, j), $0 \le l \le 1$, $0 \le i,j < n$, with a typed pixel prepares a record R = (*PixelNum*, *Type*). *PixelNum* is set to the linear number of the generated pixel, and *Type* is set to the assigned type. In step 2, $n \times n \times n$ PEs are used to move all the records on the bottom layer to the low numbered PEs inside the layer, one record per PE (observation 6).

In step 3, all the records on layer 1 are moved to the high numbered PEs inside the layer, one record per PE. These records are then moved down to the bottom layer. Since the number of U links and D links cannot be greater than n^2, we know that there is at most one record in each PE on the bottom layer. The PEs on the bottom layer which do not have a record creates a dummy record whose pixel number is n^2.

In step 4, the records on the bottom layer

are sorted in increasing order of their pixel number, *PixelNum*. At this moment, for some i, $0 \le i < n^2 - 1$, the record in PE(i) and the record in PE($i+1$) may have the same pixel with a different type. This happens when the E and W neighbors of a black pixel are both white. In such case, we delete the record in PE($i+1$) and change the *Type* in PE(i) to UD. Each typed boundary pixel $P(i,j)$ is next moved to the i-th layer of the mesh. It is then moved to the i-th row inside the layer, and then to the j-th column inside the layer, and then straight down to the bottom layer.

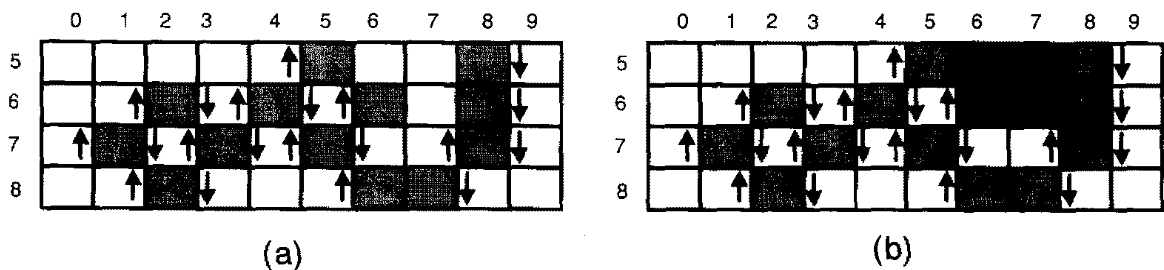
Currently, all the boundary pixels of the binary image are built and stored on the bottom layer. The remaining black pixels are generated in phase 4. This phase is the same as that in [7]. For the sake of completeness, we state the idea used in this phase. Suppose that there are k boundary pixels in row r. We now sweep row r from left to right, and remember the types of the boundary pixels we meet, in order. Suppose that t_0, t_1, t_2, ..., t_m, $m \le k$ is the sequence of the remembered types, excluding UD types. Then, even numbered types in this sequence are U's and the remaining types are D's. In other words, U is always followed by a D and the pixels in between these two must be all black. They are the pixels inside the black regions. See Fig. 5. It shows the boundary pixels generated for rows 5-8 of the image in Fig. 1. In order to generate such black pixels, each PE on the bottom layer connects its EW port, except the PE with a pixel of type D. Now the PE with a pixel of type U sends out a signal through its row bus. The PEs that received the signal generate a black pixel. In Fig. 5, $P(5,6)$, $P(5,7)$ and $P(6,7)$ are the black pixels generated in phase 4.

It is obvious that the time taken by each phase of the algorithm is $O(1)$. Thus, we can conclude the following theorem.

Theorem 2. *The above binary image generation algorithm converts a boundary code of length b to its corresponding binary image of size $n \times n$. The algorithm runs in constant time using $n \times n \times n$ PEs.*

5 Conclusion

Here, we presented efficient reconfigurable mesh algorithms for transforming between a binary image and its corresponding boundary code. These algorithms use $n \times n \times n$ processors when the size of the binary image is $n \times n$. Recent published results [9] show that these transformations can be done in $O(1)$ time using $O(n^4)$ processors. Although the algorithms run in constant time, the number of processors



(Fig. 5) (a) Boundary pixels and their types, (b) pixels generated in phase 4.

used by them does not seem to be reasonably small enough to be used in practice.

So, we focused on reducing the number of processors while keeping the execution time reasonably small. As for the algorithms for transforming between a binary image and a quadtree in [9], we used $n \times n \times n$ PEs only. Our transformation from a boundary code to a binary image takes $O(1)$ time, and the converse transformation takes $O(\log n)$ time. The time complexity of the algorithm for transforming a binary image to a boundary code is bounded by the time taken by phase 3 of the algorithm. One way to get a faster algorithm would be to make phase 3 work faster since it is the only phase which takes more than constant time.

References

[1] F. Dehne, A. G. Ferreira and A. Rau-Chaplin, "Efficient Parallel Construction and Manipulation of Quadtrees," *Proc. of International Conference on Parallel Processing*, pp.255-262, 1991.

[2] D. P. Doctor and H. Sudborough, "Efficient Parallel Sibling Finding For Quadtree Data Structure," *Proc. of Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993.

[3] Y. Hung and A. Rosenfeld, "Parallel Processing of Linear Quadtrees on a Mesh-Connected Computer," *Journal of Parallel and Distributed Computing*, vol 7, pp. 1-27, 1989.

[4] O. H. Ibarra and M. Kim, "Quadtree Building Algorithms on an SIMD Hypercube," *Journal of Parallel and Distributed Computing*, Vol 18, pp. 71-76, 1993.

[5] J. Jang and K. Kim, "A Fast Parallel Sorting Algorithm on the K-Dimensional Reconfigurable Mesh," *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, pp. 519-532, Dec. 1997.

[6] J. Jang and V. K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *Proc. of International Parallel Processing Symposium*, pp. 130-137, Mar. 1992.

[7] M. Kim, "Transformation Between a Binary Image and a Boundary Code on a Reconfigurable Mesh," *Journal of KISS (A): Computer Systems and Theory*, Vol. 25, No. 3, pp. 231-242, March 1998.

[8] M. Kim and O. H. Ibarra, "Transformations Between Boundary Codes, Run Length Codes, and Linear Quadtrees", *Proc. of International Parallel Processing Symposium*, pp. 120-125, 1994.

[9] M. Kim and J. Jang, "Constant Time Transformation Between Binary Images and Quadtrees on a Reconfigurable Mesh", *Journal of KISS (A): Computer Systems and Theory*, Vol. 23, No. 5, pp. 454-466, May 1996.

[10] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang, "Sorting in $O(1)$ time on an $n \times n$ reconfigurable mesh," *Parallel Computing: From Theory to Sound Practice, Proceedings of EWPC'92*, Plenary Address, IOS Press, Amsterdam, 1992, pp. 16-27.

[11] R. Miller, V. K. Prasanna Kumar, D. I. Reisis and Q. F. Stout, "Meshes with Reconfigurable Buses," *Proc. of Fifth MIT Conference On Advanced Research in VLSI*, pp. 163-178, 1988.

[12] R. Miller, V. K. Prasanna Kumar, D. I. Reisis and Q. F. Stout, "Data Movement Operations and Applications on Reconfigurable VLSI arrays," *Proc. International Conference on Parallel Processing* 1, pp.205-208, 1988.

[13] S. Olariu, J. L. Schwing, and J. Zhang, "Fast Computer Vision Algorithms for Rec-

onfigurable Meshes," *Proc. of International Parallel Processing Symposium*, pp. 258-261, 1992.

[14] H. Samet, *Applications of Spatial Data Structures, Computer Graphics, Image Processing, and GIS*, Addison Wesley, 1990.

[15] Q. F. Stout, "Meshes with Multiple Buses," *Proc. of IEEE Symposium on Foundations of Computer Science*, pp. 264-272, 1986.

김 명

1981년 이화여자대학교 수학과 (학사)
1983년 서울대학교 대학원 계산통계학과 (석사)
1990년 University of Minnesota, Minneapolis (석사, 박사수료)
1993년 University of California, Santa Barbara (박사)
1993년~1994년 University of California, Santa Barbara (Postdoc, 강사)
1995년~현재 이화여자대학교 컴퓨터학과 조교수
관심분야 : 병렬/분산처리, 알고리즘