선형 사진트리에서 이웃 블록을 찾기 위한 상수시간 RMESH 알고리즘

김 기 원† · 우 진 운††

요 약

사진트리를 메모리에 저장하는 방법 중 선형 사진트리 표현 방법은 다른 표현 방법과 비교할 때 저장 공간을 매우 효율적으로 절약할 수 있는 장점을 가진다. 따라서 지금까지 사진트리와 관련된 연산의 수행에 있어, 선형 사진트리를 사용하는 효율적인 알고리즘들이 많이 개발되고 연구되어 왔다. 본 논문에서는 RMESH(Reconfigurable MESH) 구조에서 3-차원 $n \times n \times n$ 프로세서를 사용하여 선형 사진트리로 표현된 이진 영상의 이웃 블록들을 찾는 알고리즘을 제안한다. 이 알고리즘은 $O(1)$시간 복잡도를 갖는다.

Constant Time RMESH Algorithm to Find Neighbor Blocks in Linear Quadtrees

Gi-Won Kim† · Jin-Woon Woo††

ABSTRACT

A linear quadtree representation as a way to store a quadtree is efficient to save space compared with other representations. It, therefore, has been widely studied to develop efficient algorithms to execute operations related with quadtrees. In this paper, we present algorithm to find neighbor blocks of binary images represented by linear quadtrees. using three-dimensional $n \times n \times n$ processors on RMESH(Reconfigurable MESH). Our algorithm have $O(1)$ time complexity.

1. 서 론

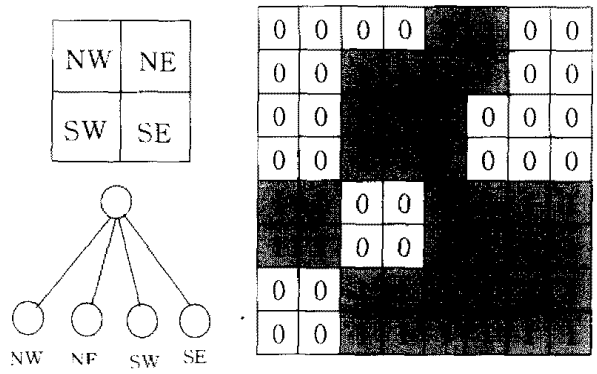
형처리, 패턴 인식 및 로보트 공학분야 등의 자료를 표현하는데 매우 적합한 기법이다. 특히 계층적 자료구조 중의 하나인 사진트리(quadtree)는 디지털 영상을 규칙적으로 분해(decomposition)하기 때문에 이진영상을 표현하는데 매우 유용한 자료구조이다[1, 2, 3].

※ 본 연구는 단국대학교의 교내연구비 지원에 의하여 연구되었음.
† 준 회 원 : 단국대학교 대학원 전산통계학과
†† 종신회원 : 단국대학교 전산통계학과 교수
논문접수 : 1998년 9월 16일, 심사완료 : 1998년 11월 4일

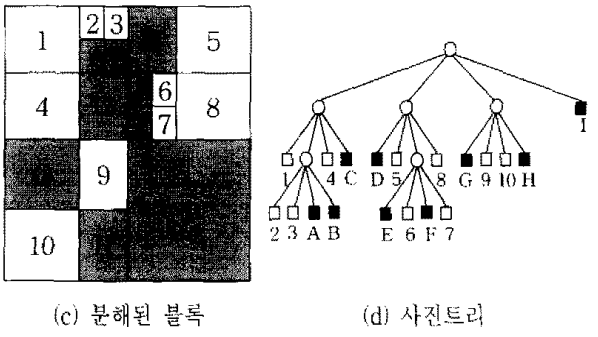
$n \times n$ 이진 영상, ($n=2^k$, k는 양의 정수)에 대한 사진트리는 다음과 같이 정의된다. 사진트리의 루트(root) 노드는 전체 영상을 표현하는 것으로, 만약 영상의 모든 픽셀(pixel)들이 같은 색을 가진다면 루트 노드는 자식 노드를 갖지 않지만, 서로 다른 색을 가진다면 루트 노드는 4 개의 자식 노드를 갖는다. 자식 노드는 왼쪽부터 각각 영상의 NW, NE, SW 및 SE 블록(block)의 색을 표현한다(그림 1(a) 참조). 이와 같은 분해 과정은 노드가 표현하는 블록이 단지 하나의 공통된 색을 가지게 될 때까지 4 개의 자식 노드에 대

해 순환적으로 적용된다.

예를 들면, 8×8 이진 영상을 사진트리로 표현해 보자. 일반적으로 이진 영상에서는, 그림 1(b)와 같이 WHITE는 0으로 BLACK은 1로 표현한다. 그림 1(c)는 그림 1(b)를 분해한 최종 결과를 블록으로 나타낸 것이고, 그림 1(d)는 그림 1(c)의 블록에 대해 사진트리로 표현한 것이다. 그림 1(c)와 그림 1(d)에서 WHITE 블록은 숫자, BLACK 블록은 영문자로 구별하였다. 그림 1(d)에서 사각형 BLACK 노드는 블록 전체가 1로 구성되어 있음을 의미하며, 사각형 WHITE 노드는 블록 전체가 0으로 구성되어 있음을 의미한다. 그리고 원형 노드는 내부 노드로서 GRAY 노드라 한다.



(a) 블록과 노드와의 관계 (b) 8×8 이진영상



(c) 분해된 블록 (d) 사진트리

(그림 1) 이진 영상과 사진트리와의 관계
(Fig. 1) Relationship between a binary image and its quadtree

사진트리에서 레벨(level)은 루트 노드에서 임의의 노드까지의 거리로 정의하며, 루트 노드의 레벨은 0으로 한다. 그리고 사진트리의 높이는 $\log_4(n\times n)$ 값으로 정의한다. 사진트리의 높이가 h일 때 레벨이 l인 노드의 블록 크기를 $2^{(h-l)}\times 2^{(h-l)}$로 계산할 수 있다. 다시 말해서 이 블록은 $4^{(h-l)}$개의 픽셀들의 모임을 표현한다. 예를 들면, 그림 1(d)에서 노드 I는 4×4, 노드 D는 2×2, 노드 E는 1×1의 블록 크기를 갖는다.

지금까지 사진트리를 메모리에 저장하기 위한 여러 가지 방법들이 제안되었다. 그 중 트리 구조를 사용하는 방법은 각 노드가 자신의 자식 노드를 가르키는 포인터 값을 저장하는 공간을 필요로 하므로 사진트리를 구성하는 노드들의 수가 많을 경우 포인터를 기억하기 위한 많은 저장 공간을 필요로 하는 단점이 있다. 이러한 단점을 보완하기 위하여 선형 사진트리(linear quadtree) 표현 방법을 사용한다[1].

선형 사진트리 표현 방법은 사진트리의 BLACK 노드에 해당되는 블록의 위치와 크기에 관한 정보, 즉 (*Index*, *Level*)만을 저장하는 것이다. 이때 (*Index*, *Level*)을 위치 코드(locational code)라 한다. 여기에서 *Index*는 사진트리 노드에 해당하는 블록의 맨위 왼쪽에 있는 픽셀의 shuffled row-major 인덱스이다.

$n=2^i$, $i>0$인 $n\times n$ 이진 영상의 픽셀에 인덱스를 부여하는 방법은 여러 가지가 있으나, 그 중 가장 널리 사용되는 방법은 row-major 인덱스, column-major 인덱스, shuffled row-major 인덱스이다. Row-major 인덱스 방법은 r행과 c열의 픽셀에 $r\times n+c$의 인덱스를 부여하고, column-major 인덱스 방법은 r행과 c열의 픽셀에 $c\times n+r$의 인덱스를 부여하며, shuffled row-major 인덱스 방법은 r과 c의 이진 표현이 $r_{i-1}\cdots r_1 r_0$과 $c_{i-1}\cdots c_1 c_0$, $(i=\log_2 n)$ 일 때, 해당 픽셀에 이진수 표현으로 $r_{i-1}c_{i-1}\cdots r_1 c_1 r_0 c_0$의 인덱스를 부여한다. 따라서 이러한 2가지 인덱스 사이의 상호 변환이 가능하며, 인덱스의 위치를 나타내는 행과 열 번호도 쉽게 구할 수 있다.

그림 1(b)의 8×8 이진 영상에 shuffled row-major 인덱스를 부여하면 그림 2(a)와 같고, 그림 1(d)의 BLACK 노드들을 위치 코드를 이용하여 선형 사진트리 표현으로 나타내면 그림 2(b)와 같다.

그림 2(b)와 같이 선형 사진트리의 표현에서 WHITE 노드에 대한 정보는 저장하지 않고 BLACK 노드에 대한 정보만을 저장하는 이유는 사진트리를 다시 구축하지 않고도 BLACK 노드에 대한 정보를 이용하여 WHITE 노드에 대한 정보를 쉽게 구할 수

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

(a) 픽셀에 부여된 shuffled-row major 인덱스

BLACK

노드	: A	B	C	D	E	F	G	H	I
Index	: 6	7	12	16	24	26	32	44	48
Level	: 3	3	2	2	3	3	2	2	1

(b) 선형사진트리

(그림 2) 위치 코드를 이용한 선형 사진트리 표현
(Fig. 2) The linear quadtree representation using locational code

있으며, 또한 BLACK 노드에 대한 정보만을 저장하므로써 저장 공간을 최소화할 수 있는 장점이 있기 때문이다.

이웃 블록 찾기는 영상 처리의 응용에서 중요하게 사용되는 기하학적 성질에 속한다. 병렬 컴퓨터와 관련하여 Hung과 Rosenfeld는 $n\times n$ mesh 구조에서 $n\times n$ 이진 영상의 이웃 찾기를 위한 $O(n)$ 시간 알고리즘을 제안하였다[4]. 본 논문에서는 이진 영상의 이웃 블록 찾기를 위한 상수시간 알고리즘을 제안하며, 이 알고리즘은 $n\times n\times n$ RMESH에서 $O(1)$ 시간 복잡도를 갖는다.

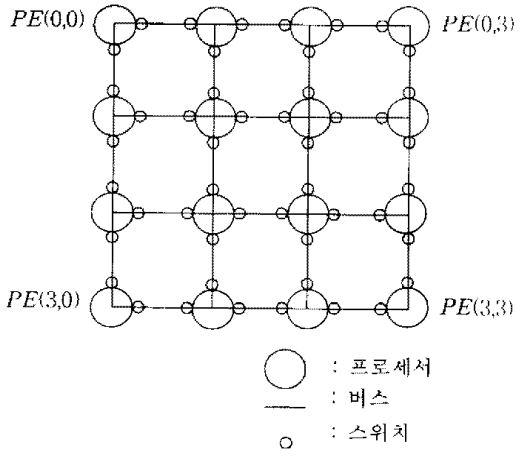
지금까지 $n\times n\times n$ RMESH 상에서 상수 시간을 갖는 많은 이진 영상 알고리즘들이 개발되었는데, 이진 영상과 선형 사진트리 사이의 상호 변환 알고리즘[5], 선형 사진트리의 집합 연산[6], 선형 사진트리의 면적과 둘레 길이 계산[7] 등을 들 수 있다. 본 논문에서 제안하는 알고리즘은 [4]의 알고리즘에 비하여 더 많은 프로세서들을 사용하지만 상수 시간의 시간 복잡도를 가지므로 시간적인 면에서 매우 효율적이라 할 수 있다.

2. RMESH 구조

RMESH는 Reconfigurable MESH의 약어로서 기존의 메쉬(mesh) 구조에 동적으로 재구성 가능한 버스 시스템을 결합한 구조이다. 이 구조는 Miller, Prasanna-Kumar, Reisis, Stout에 의하여 처음 제안되었으며[8], 구조적인 장점 때문에 다양한 분야에서 연구되었고 효율적인 알고리즘들이 개발되었다[9, 10, 11]. 또한 버스 시스템의 재구성 방법 면에서 서로 차이를 갖는 PARBUS 구조와 MRN 구조가 제안되었다[12, 13].

2.1 2-차원 RMESH

크기가 $n\times n$인 2-차원 RMESH의 기본 구조는 메쉬이며 프로세서들 사이의 통신을 위하여 브로드캐스트 버스(broadcast bus)가 존재한다. 예를 들어, 그림 3은 4×4 RMESH 구조를 보여준다. 프로세서들을 식별하기 위해 각 프로세서에게 $PE(i,j)$ 를 부여한다. 이때 $0\le i,j<n$, i는 행의 인덱스이고, j는 열의 인덱스이다.



(그림 3) 4×4 RMESH 구조
(Fig. 3) 4×4 RMESH structure

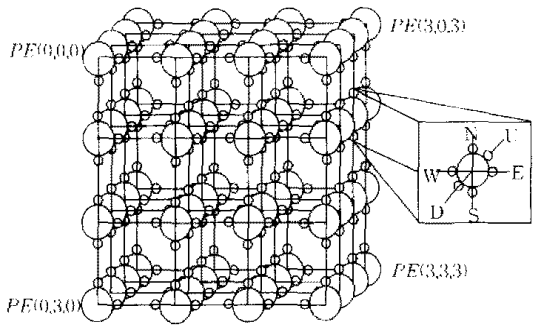
브로드캐스트 버스상의 통신 제어를 위하여 버스 스위치가 있다. 버스 스위치들은 각 프로세서의 상, 하, 좌, 우에 하나씩 존재하는데, 이를 각각 N(north), S(south), W(west), E(east)라 한다. 버스 스위치는 각 프로세서의 소프트웨어에 의하여 $O(1)$ 시간에 조작되며, 스위치의 개폐 여부에 따라 브로드캐스트 버스를 다수의 서브버스(subbus)들로 재구성이 가능하다. 예를

놓여, 각 프로세서가 자신의 S와 N 스위치를 끊고 E와 W 스위치를 연결한다면 여러 개의 서브버스가 형성되는데, 이를 행 버스(row bus)라 하고, 자신의 E와 W 스위치를 끊고 S와 N 스위치를 연결한다면 여러 개의 서브버스가 형성되는데, 이를 열 버스(column bus)라 한다.

두 개의 프로세서들은 충돌이 없는 한 공통된 하나의 특정 스위치를 동시에 개폐할 수 있다. 버스상에는 특정 시간에 단 하나의 프로세서만이 데이터를 실을 수 있으며, 서브버스 위에 실린 데이터는 단위 시간에 그 버스에 연결된 모든 프로세서에게 전달될 수 있다. 만약 한 프로세서가 서브버스상에 있는 모든 프로세서에게 레지스터(register) X의 값을 브로드캐스트하려면 broadcast(X) 명령을 사용하고, 브로드캐스트 버스의 내용을 읽어 레지스터 R에 저장하려면 R := content (broadcast bus) 명령을 사용한다. 따라서 데이터 브로드캐스트는 $O(1)$ 시간에 수행된다.

2.2 3-차원 RMESH

2-차원 RMESH를 확장하여 3-차원 RMESH를 구성할 수 있다. 3 차원 RMESH에서는 각 프로세서에게 $PE(l, i, j)$를 부여한다. 이때 $0 \le l, i, j < n$. l은 각 프로세서가 위치한 계층(layer)이고, i와 j는 계층 l에서의 행과 열의 인덱스이다. 예를 들어, 그림 4는 $4 \times 4 \times 4$ RMESH를 보여준다. 버스 스위치들은 기본적으로 2 차원 RMESH와 같이 N, S, W, E 스위치가 존재하며, 추가적으로 각 프로세서마다 계층을 연결하는 U(up)와 D(down) 스위치가 존재한다. 그리고 모든 프로세서의 N, S, W, E 스위치를 끊고 U와 D 스위치를 연결하면 여러 개의 서브버스가 형성되는데, 이를 UD 버스라 한다.



(그림 4) $4 \times 4 \times 4$ RMESH 구조
(Fig. 4) $4 \times 4 \times 4$ RMESH structure

3. 기본적인 연산

여기서는 3차원 RMESH 구조에서 유용한 기본적인 연산들을 알아 보며, 이 알고리즘들에 대한 설명은 17] 에 제시된다.

3.1 3차원 RMESH의 재구성

3차원 $n \times n \times n$ RMESH 구조를 $n \times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다. 이 연산은 계층 0에 row-major 순서로 저장된 위치 코드들을 일련의 연속된 위치 코드들로 재구성하기 위해 사용되며, 계층 0의 행 i에 있는 위치 코드들을 계층 i의 행 0으로 이동시킨 후, 홀수 번째 계층에 있는 위치 코드를 역순으로 permute함으로써 만들어진다. 이 연산은 $O(1)$ 시간에 수행 가능하다.

3.2 위치 코드의 분해

이 연산은 크기가 $s \times s$ $(1 < s \le n)$인 블록을 나타내는 위치 코드를 그 블록에 포함된 1×1 블록을 나타내는 위치 코드들로 분해하는 것으로, $O(1)$ 시간에 수행 가능하다. 예를 들어, 높이가 2인 사진 트리에서 위치 코드 (12, 1)은 2×2 크기의 블럭이므로 이 위치 코드는 (12, 2), (13, 2), (14, 2), (15, 2) 라는 4개의 위치 코드들로 분해되어야 한다.

3.3 위치 코드의 이동

위치코드들이 $n \times n \times n$ RMESH의 계층 0에 속하는 프로세서에 하나씩 저장되어 있을 때, 위치 코드가 나타내는 *Index* 값을 row major 인덱스의 행과 열 번호 (i, j)로 변환한 후, 이 위치 코드를 계층 0의 프로세서 $PE(0, i, j)$로 이동하는 연산으로 $O(1)$ 시간에 수행 가능하다.

3.4 정 렬

Nigam과 Sahni[14]는 rotate sort 알고리즘을 3-차원 $n \times n \times n$ RMESH에 적용하여 n^2 개의 데이터를 $O(1)$ 시간에 정렬할 수 있는 알고리즘을 제안하였다. 이 알고리즘에서 초기의 n^2 개의 데이터는 계층 0에 속하는 $PE(0, i, j)$ $(0 \le i, j < n)$에 존재하며, 정렬된 결과는 계층 0에 속하는 프로세서에 row-major 순서

로 하나씩 저장된다. 즉, $PE(0,0,0)$, $PE(0,0,1)$, $\cdots$, $PE(0,1,0)$, $PE(0,1,1)$, $\cdots$, $PE(0,n-1,n-1)$의 순서로 저장된다.

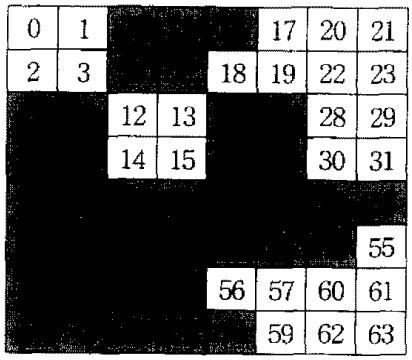
4. 이웃 블록 찾기

이웃 블록 찾기는 사진트리의 각 블록에 대하여 동, 서, 남, 북쪽에 이웃한 블록들을 구하는 것이다[4]. 동쪽에 인접한 이웃 블록을 찾는 예를 살펴보자. 그림 5(a)에 이진 영상과 위치 코드가 주어진다. 위치 코드 (4, 2)가 나타내는 블록은 동쪽 이웃으로 위치 코드 (16, 3)이 나타내는 블록을 가지는데, 이것을 단순히 위치 코드 (4, 2)의 동쪽 이웃은 위치 코드 (16, 3)이 된다고 말한다. 그리고 위치 코드 (32, 1)의 동쪽 이웃은 (48, 2), (58, 3)이 되며, 이웃 블록들은 모두 기준 블록보다 크기가 작다. 한편 위치 코드 (48, 2)의 서쪽 이웃은 (32, 1)이 되며, 이웃 블록이 기준 블록보다 크다. 여기서 (32, 1)과 (48, 2) 사이의 관계에서 보듯이, 어떤 블록 A의 작은 크기의 동쪽 이웃 블록이 B이면, 역으로 블록 A는 블록 B의 큰 크기의 서쪽 이웃 블록이 된다는 것을 알 수 있다. 따라서 각 방향의 작은 크기의 이웃 블록들만을 알고 있다면 반대 방향의 큰 크기의 이웃 블록을 찾을 수 있다.

이웃 블록을 쉽게 나타내는 한 가지 방법은 위치 코드가 다른 어떤 위치 코드의 이웃인가를 나타내는 것으로, 이것을 기준 위치 코드라 한다. 위의 예에서 (16, 3)은 (4, 2)의 동쪽 이웃이 되므로 (16, 3)의 기준 위치 코드는 (4, 2)가 된다. 그림 5(a)의 위치 코드들의 기준 위치 코드를 구하면 그림 5(b)와 같게 되며, '-' 표시는 기준 위치 코드가 없음을 나타낸다. 그리고 각 블록의 실제 이웃 블록들을 알고자 한다면 그림 5(b)의 기준 위치 코드를 기준으로 정렬하여 구할 수 있으며, 그림 5(c)는 그 결과를 보여준다. 여기서 (32, 1)의 동쪽 이웃 블록은 (48, 2)와 (58, 3)이 됨을 알 수 있다.

또한 네 방향의 이웃 블록을 구할 때, 기준 블록에 대해 각 방향의 이웃 블록들은 서로 독립적이므로 각 방향의 이웃 블록들을 독립적으로 찾는 것이 가능하다. 그리고 한 방향의 이웃 블록 찾기 알고리즘을 이용하여 방향만 변경한다면 다른 방향에 대해서 쉽게 적용된다.

블록 A는 크기가 같거나 작은 동쪽 이웃블록을 갖는다고 하자. 그리고 블록 B는 맨 위의 동쪽에 이웃한



Index : 4 8 16 24 32 48 52 53 54 58
Level : 2 2 3 2 1 2 3 3 3 3

(a) 이진 영상과 위치코드

위치 코드	기준 위치 코드
(4, 2)	-
(8, 2)	-
(16, 3)	(4, 2)
(24, 2)	-
(32, 1)	-
(48, 2)	(32, 1)
(52, 3)	(48, 2)
(53, 3)	(52, 3)
(54, 3)	(48, 2)
(58, 3)	(32, 1)

(b) 기준 위치 코드

위치 코드	동쪽 이웃
(4, 2)	(16, 3)
(32, 1)	(48, 2) (58, 3)
(48, 2)	(52, 3) (54, 3)
(52, 3)	(53, 3)

(c) 동쪽 이웃의 위치 코드

(그림 5) 이웃 찾기 알고리즘 적용 예
(Fig. 5) An example to find neighbor blocks

블록이라 가정하자(이것은 A와 B의 행(row) 번호가 같다는 것을 의미하며, 본 논문에서는 블록 B가 항상 존재한다고 가정한다). 그 다음에 블록들을 (row,col)을 기준으로 올림차순으로 정렬하면, 각 블록을 나타내는 대표값은 (row, col) 순서가 된다. 이러한 과정에 의해, 블록 A는 정렬 리스트의 블록 B 바로 앞에 위치하게 된다. 그 후에 블록 B는 블록 A로부터 관련된 정보를 쉽게 얻을 수 있다. 만약, 다른 이웃 블록이 존재한다면, A의 다른 동쪽 이웃 블록들에게 그

정보를 전달해야 한다. 이것은 A의 모든 동쪽 이웃 블록들은 같은 열(column) 번호를 갖는 성질을 이용하여 쉽게 해결할 수 있다. 먼저 블록들을 (col, row)에 따라 정렬한다. 그러면 B는 정렬된 리스트에 있는 A의 동쪽 이웃 블록들 중 맨 처음에 위치하게 된다. 따라서 B가 가진 정보를 다른 동쪽 블록들에게 전달할 수 있게 된다.

여기서는 크기가 같거나 작은 동쪽 이웃 블록을 찾는 알고리즘만을 설명하기로 한다. 그리고 이 알고리즘을 약간 수정하여 다른 방향의 이웃 블록들을 찾을 수 있으며, 앞에서 언급하였듯이 이 알고리즘의 결과를 이용하여 큰 이웃 블록을 찾을 수도 있다.

초기에 k ($0<k\leq n^2$) 개의 위치코드는 계층 0에 속하는 k 개의 프로세서에 row-major 순서로 하나씩 저장되어 있다고 가정한다. 이웃 찾기를 위한 RMESH 알고리즘은 다음과 같이 11 단계로 구성된다.

[단계 1] : 위치 코드 <*Index*, *Level*>를 갖는 계층 0의 프로세서는 *Index* 값의 row-major 인덱스와 column-major 인덱스를 계산하여 *RMAdd*와 *CMAdd* 필드에 저장한다. 그리고 이 인덱스와 블록의 크기를 이용하여 크기가 작거나 같은 동쪽 이웃 블록이 가질 수 있는 column major 인덱스의 시작 인덱스와 끝 인덱스를 계산하여 *Begin*과 *End*에 저장한다. 계산된 필드들을 위치 코드에 추가하여 <*Index, Level, RMAdd, CMAdd, Begin, End*>를 새로운 위치 코드로 만든다.

[단계 2] : 위치 코드의 *RMAdd*를 기준으로 올림차순으로 정렬한다.

[단계 3] : 계층 0의 행 i에 있는 위치코드들을 계층 i의 행 0으로 이동시킨 후, 홀수번째 계층에 있는 위치코드를 역순으로 permute한다. 그리고 $n\times n\times n$ RMESH를 $n\times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다.

[단계 4] : 왼쪽 프로세서의 위치 코드를 받아 들여 *Index, Begin, End* 만을 *NIndex, NBegin, NEnd* 필드에 저장한다. 자신의 *RMAdd*를 이용하여 자신이 인접한 블록의 최상위 이웃 블록인가를 확인하여 이에 해당하면 *Top* 필드에 1을 저장하고 그렇지 않으면 0을 저장한다. 그 다음, *Top, NIndex, NBegin, NEnd* 필드를 위치 코드에 추가한다.

[단계 5] : 위치 코드를 계층 0의 프로세서에 row-major 순서로 하나씩 저장한다.

[단계 6] : 위치 코드의 *CMAdd*를 기준으로 올림차순으로 정렬한다.

[단계 7] : 계층 0의 행 i에 있는 위치코드들을 계층 i의 행 0으로 이동시킨 후, 홀수번째 계층에 있는 위치코드를 역순으로 permute한다. 그리고 $n\times n\times n$ RMESH를 $n\times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다.

[단계 8] : *Top*의 값이 1인 프로세서는 왼쪽 스위치를 끊음으로써 세그먼트를 만드는데, 이 프로세서는 행 버스를 이용하여 자신의 위치 코드를 세그먼트 내의 다른 프로세서들에게 브로드캐스팅한다.

[단계 9] : 각 프로세서는 전달받은 위치 코드에서 *NIndex, NBegin, NEnd* 값만을 *RIndex, RBegin, REnd* 레지스터에 저장한다. 자신의 *CMAdd*를 이용하여 자신도 이웃 블록인가를 확인하여 이에 해당하면 *Neighbor* 필드에 *RIndex*를 저장하고 그렇지 않으면 ∞를 저장한 후, 이 필드를 위치 코드에 추가한다.

[단계 10] : 위치코드들을 계층 0의 프로세서에 row-major 순서로 하나씩 저장한다.

[단계 11] : 위치 코드의 *Index*를 기준으로 올림차순으로 정렬한다.

이진 영상에 대한 이웃 블록 찾기를 수행하는 RMESH 알고리즘이 실행되는 과정을 단계별로 살펴보자. $n\times n$ 이진영상에 대하여 3-차원 $n\times n\times n$ RMESH 구조를 사용한다. 그리고 초기에 선형 사진트리로 표현된 위치 코드들이 $n\times n\times n$ RMESH의 계층 0에 속하는 프로세서에 row-major 순서로 하나씩 저장되어 있다고 가정한다.

[단계 1]에서 위치 코드의 *Index*는 shuffled row-major 인덱스인데 이를 다른 인덱스로 바꾸기 위해서, *Index*를 2진수들로 표현하여 2진수의 홀수번째 비트들을 선택해서 행 번호를 얻고, 짝수번째 비트들을 선택해서 열 번호를 얻는다. 행 번호와 열 번호를 이용하

여 row-major 인덱스와 column-major 인덱스를 계산하여 *RMAdd*와 *CMAdd* 필드에 저장한다. 예를 들어, 그림 5(a)에서 *Index*가 32인 위치 코드의 경우, 32의 2진수 표현이 100000 이므로, 행 번호는 100 즉 십진수 4가 되고 열 번호는 000 즉 십진수 0이 된다. 행 번호와 열 번호를 이용하여 row major 인덱스를 구하면 $4\times8+0=32$가 되고, column-major 인덱스를 구하면 $0\times8+4=4$가 된다.

이 위치 코드가 나타내는 블록에 대해 크기가 같거나 작은 동쪽 이웃 블록이 가질 수 있는 column-major 인덱스의 시작 인덱스 *Begin*과 끝 인덱스 *End*를 다음과 같이 계산한다.

$$RMSucc = RMAdd + \sqrt{size},$$
$$size = 2^{height-level} \times 2^{height-level}$$
$$Begin = RMSucc\text{의 column-major 인덱스}$$
$$End = Begin + \sqrt{size} - 1$$

위에서 *RMSucc*는 row-major 인덱스이므로 행 번호와 열 번호를 구하여 column-major 인덱스를 얻을 수 있다. 지금까지 행하여진 연산들은 모두 단순 계산들이므로 단계 1의 수행 시간은 $O(1)$이다.

이러한 계산 과정을 예로 들어 보자. 그림 5(a)에서 (32, 1)의 경우, *Index*가 32이므로 *RMAdd* = 32, *CMAdd* = 4 가 된다. 이므로 $RMSucc = 32 + 2^2 = 36$. 그리고 $36 = 4\times8+4$ 이므로 *RMSucc*의 행번호와 열 번호는 각각 4, 4이다. 따라서 *RMSucc*의 column-major인덱스는 $4\times8+4=36$이 되므로 *Begin* = 36 이고 $End = 36 + 2^2 - 1 = 39$ 가 된다. 이와같이 각 위치 코드에 대해 *Begin*과 *End* 값을 구하면 그림 6과 같게된다.

Index:	4	8	16	24	32	48	52	53	54	58
Level:	2	2	3	2	1	2	3	3	3	3
RMAdd:	2	16	4	20	32	36	38	39	46	60
CMAdd:	16	2	32	34	4	36	52	60	53	39
Begin:	32	18	40	50	48	52	53	5	61	47
End:	33	19	40	51	58	53	53	5	61	47

(그림 6) 단계 1의 결과
(Fig. 6) The result of step 1

[단계 2]는 계층 0의 프로세서에 저장된 위치 코드를 *RMAdd* 필드를 기준으로 오름차순으로 정렬하여 위치 코드를 row-major 순서로 프로세서에 할당한다. 이 단계는 정렬로서 앞에서 언급한 정렬 알고리즘을 적용하면 $O(1)$ 시간에 수행된다.

[단계 3]은 기본적인 연산으로 $n\times n\times n$ RMESH를 $n\times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다. 이 과정을 통하여 각 계층의 프로세서들은 인접한 계층의 프로세서들과 맨 좌측, 혹은 맨 우측에서 연결된 형태로 재구성된다. 이러한 구성은 n^2 개의 프로세서들이 n번 연속적으로 연결된 형태이며, 이 단계는 $O(1)$ 시간에 수행 가능하다.

[단계 4]에서 각 프로세서는 왼쪽 프로세서의 위치 코드를 받아들여, 자신이 인접한 블록들 중 최상위 블록인가를 확인한다. 이를 위해 왼쪽 프로세서의 위치 코드 중 *Index, Begin, End* 값을 *NIndex, NBegin, NEnd* 필드에 저장한 후, 다음과 같은 조건을 만족하면 *Top* 필드를 1로 하고, 그렇지 않으면 *Top* 필드를 0으로 한다.

$$NBegin \le CMAdd \text{ 이고 } CMAdd + \sqrt{size} - 1 \le NEnd,$$
$$size = 2^{height-level} \times 2^{hairht-level}$$

이와같이 결정된 *Top* 필드와 더불어 *NIndex, NBegin, NEnd* 필드를 위치 코드에 추가한다. 이 단계에서는 인접한 왼쪽 프로세서만을 접근하고 단순계산만을 수행하므로 $O(1)$ 시간에 수행 가능하다.

Index:	4	16	8	24	32	48	52	53	54	58
Level:	2	3	2	2	1	2	3	3	3	3
RMAdd:	2	4	16	20	32	36	38	39	46	60
CMAdd:	16	32	2	34	4	36	52	60	53	39
Begin:	32	40	18	50	48	52	53	5	61	47
End:	33	40	19	51	58	53	53	5	61	47
NIndex:	∞	4	16	8	24	32	48	52	53	54
NBegin:	∞	32	40	18	50	48	52	53	5	61
NEnd:	∞	33	40	19	51	58	53	53	5	61
Top:	0	1	0	0	0	1	1	1	0	0

(그림 7) 단계 2에서 단계 4까지의 결과
(Fig. 7) The result of steps 2 to 4

[단계 5]는 위치 코드들을 계층 0의 프로세서에 row-major 순서로 저장하는 단계로서, 먼저 계층 i의 행 0에 있는 위치코드들을 계층 0의 행 i로 이동시킨다. 이 과정은 단계 3의 과정을 역순으로 수행할 수 있으며, 소요 시간은 $O(1)$이다.

[단계 6]은 계층 0의 프로세서에 저장된 위치 코드를 CMAdd 필드를 기준으로 오름차순으로 정렬하여 위치 코드를 row-major 순서로 프로세서에 할당한다. 이 단계는 정렬로서 앞에서 언급한 정렬 알고리즘을 적용하면 $O(1)$ 시간에 수행된다.

[단계 7]에서는 단계 3과 동일한 방법으로 $n\times n\times n$ RMESH를 $n\times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다. 단계 3과 마찬가지로 $O(1)$ 시간에 수행 가능하다.

[단계 8]에서는 *Top*의 값이 1인 프로세서는 왼쪽 스위치를 끊음으로써 세그먼트를 만드는데, 이 프로세서는 행 버스를 이용하여 자신의 위치 코드를 세그먼트 내의 다른 프로세서들에게 브로드캐스팅한다. 같은 세그먼트 내에서 브로드캐스팅만 일어나므로 이 단계는 $O(1)$ 시간에 수행 가능하다.

[단계 9]에서 각 프로세서는 전달받은 위치 코드에서 *NIndex*, *NBegin*, *NEnd* 값만을 *RIndex*, *RBegin*, *REnd* 레지스터에 저장한다. 자신의 *CMAdd*를 이용하여 자신도 이웃 블록인가를 다음과 같이 확인한다.

$RBegin \le CMAdd$ 이고 $CMAdd+\sqrt{size}-1 \le REnd$,
$size = 2^{height-level} \times 2^{height-level}$

이에 해당하면 *Neighbor* 필드에 *RIndex*를 저장하고 그렇지 않으면 ∞를 저장한 후, 이 필드를 위치 코드에 추가한다. 그림 7이 단계 9까지 수행되면 그림 8과 같게 된다.

Index:	8	32	4	16	24	48	58	52	54	53
Level:	2	1	2	3	2	2	3	3	3	3
RMAdd:	16	32	2	4	20	36	60	38	46	39
CMAdd:	2	4	16	32	34	36	39	52	53	60
Begin:	18	48	32	40	50	52	47	60	61	5
End:	19	58	33	40	51	53	47	60	61	5
NIndex:	16	24	∞	4	8	32	54	48	53	52
NBegin:	40	50	∞	32	18	48	61	52	5	60
NEnd:	40	51	∞	33	19	58	61	53	5	60
Top:	0	0	0	1	0	1	0	1	0	1
RIndex:	∞	∞	∞	4	4	32	32	48	48	52
RBegin:	∞	∞	∞	32	32	48	48	52	52	53
REnd:	∞	∞	∞	33	33	58	58	53	53	53
Neighbor:	∞	∞	∞	4	∞	32	32	48	48	52

(그림 8) 단계 5에서 단계 9까지의 결과
(Fig. 8) The result of steps 5 to 9

[단계 10]에서는 위치코드들을 계층 0의 프로세서에 row-major 순서로 저장하는 단계로서, 먼저 계층 i의 행 0에 있는 위치코드들을 계층 0의 행 i로 이동시킨다. 이 과정은 단계 7의 과정을 역순으로 수행할 수 있으며, 소요 시간은 $O(1)$이다.

[단계 11]은 계층 0의 프로세서에 저장된 위치 코드를 *Index* 필드를 기준으로 오름차순으로 정렬하여 위치 코드를 row-major 순서로 프로세서에 할당한다. 이 단계는 정렬로서 앞에서 언급한 정렬 알고리즘을 적용하면 O(1) 시간에 수행된다. 그림 8의 위치 코드들을 정렬한 결과는 그림 9와 같다.

Index:	4	8	16	24	32	48	52	53	54	58
Level:	2	2	3	2	1	2	3	3	3	3
(기준 위치 코드)										
Neighbor:	∞	∞	4	∞	∞	32	48	52	48	32

(그림 9) 단계 11 후의 기준 위치 코드
(Fig. 9) The locational code after step 11

지금까지 이웃 블록 찾기 알고리즘의 각 단계가 모두 $O(1)$ 시간에 수행될 수 있음을 설명하였으며, 다음과 같이 요약할 수 있다.

정리 k ($0<k\le n^2$) 개의 위치 코드가 $n\times n\times n$ RMESH의 계층 0에 row-major 순서로 각 프로세서에 하나씩 저장되어 있을 때, 이웃 블록 찾기 알고리즘은 $O(1)$ 시간에 수행된다.

5. 결 론

본 논문에서는 3-차원 $n\times n\times n$ RMESH 구조에서 선형 사진트리로 표현된 이진 영상의 이웃 블록 찾기를 위한 상수 시간 알고리즘을 제안하였다. 이 알고리

[illegible] 제안한 RMESH 구조에서 유용한 기본적인 연산들을 사용하였으며, 이 연산들은 모두 $O(1)$ 상수 시간 복잡도를 가진다.

참 고 문 헌

[1] H. Samet, Application of Spatial Data Structures, Computer Graphics, Image Processing, and GIS. Addison-Wesley, 1990.

[2] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, 1990.

[3] H. Samet and M. Tamminen, "Computing Geometric Properties of Images Represented by Linear Quadtrees," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.PAMI-7, No.2, March 1985.

[4] 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.

[5] 김명, 장주욱, "재구성가능 매쉬에서 $O(1)$ 시간 복잡도를 갖는 이진영상/사진트리 변환 알고리즘," 정보과학회논문지(A), 제23권, 제5호, pp.454-466, 1996.

[6] 공헌택, 우진운, "RMESH 구조에서 선형사진트리의 집합 연산을 위한 상수 시간 알고리즘," 정보과학회 논문지(A), 제24권, 제11호, pp.1218-1231, 1997.

[7] 김기원, 우진운, "선형 사진트리로 표현된 이진 영상의 면적과 둘레 길이를 계산하기 위한 상수 시간 RMESH 알고리즘," 정보처리 논문지, 제5권, 제7호, pp.1746-1758, 1998.

[8] R. Miller, V. Prasanna-Kumar, D. Reisis, and Q. Stout, "Parallel Computation on Reconfigurable Meshes," IEEE Transactions on Computers, Vol. 42, No.6, pp.678-692, 1993.

[9] J. Jenq and S. Sahni, "Reconfigurable Mesh Algorithms for The Hough Transform," Proceedings of International Conference on Parallel Processing, Vol.III, pp.34-41, 1991.

[10] J. Jenq and S. Sahni, "Reconfigurable Mesh Algorithms for Image Shrinking, Expanding, Clustering, and Template Matching," Proceedings 5th International Parallel Processing Symposium, pp.208-215, 1991.

[11] 김홍근, 조유근, "단순다각형의 내부점 가시도를 위한 효율적인 RMESH 알고리즘," 정보학회 논문지, 제20권 11호, pp.1693-1701, 1993.

[12] J. Jang, H. Park, and V. Prasanna, "A Fast Algorithm for Computing Histogram on a Reconfigurable Mesh," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.17, No.2, pp.97-106, 1995.

[13] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, "The Power of Reconfiguration," Journal of Parallel and Distributed Computing, 13, pp.139-153, 1991.

[14] M. Nigam and S. Sahni, "Sorting n Numbers On $n \times n$ Reconfigurable Meshes With Buses," Proceedings 7th International Parallel Processing Symposium, pp.174-181, 1993.

김 기 원

e-mail : gwkim@ns.dankook.ac.kr

1988년 단국대학교 계산통계학과 (학사)

1992년 단국대학교 전산통계학과 (석사)

1994년~현재 단국대학교 전산통계학과 박사 과정(수료)

관심분야 : 알고리즘, 병렬 처리, 병렬 알고리즘

우 진 운

e-mail : jwwoo@ns.dankook.ac.kr

1980년 서울대학교 수학교육과(학사)

1989년 미국 University of Minnesota 전산학과 (박사)

1980년~1983년 대한항공 및 국토개발연구원 전산실 근무

1989년~현재 단국대학교 전산통계학과 부교수

관심분야 : 알고리즘, 병렬 처리, 병렬 알고리즘