PARALLEL ALGORITHMS AND ARCHITECTURES FOR BINARY IMAGE COMPONENT LABELING

Quoc Tuan Pham

August 1987
CSD-870041

UNIVERSITY OF CALIFORNIA

LOS ANGELES

Parallel Algorithms and Architectures for Binary Image Component Labeling

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

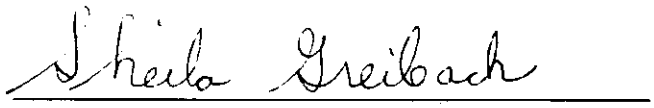
Quoc Tuan Pham

1987

The thesis of Quoc Tuan Pham is approved.

Jack Carlyle

Eli Gafni



Sheila Greibach, Committee Chair

University of California, Los Angeles

1987

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

FIGURE PAGE

ACKNOWLEDGMENTS

I would like to thank Professor Greibach for her invaluable thesis guidance and advices, Professor Carlyle and Professor Gafni for serving in my thesis committee. I would like also to thank Dr. Ashoke Deb of Memorial University and Ching-Tsun Chou for a number of improvements in my thesis.

I would like to thank the Computer Science Department for the generous financial assistance I received during my Master's program at UCLA.

ABSTRACT OF THE THESIS

Parallel Algorithms and Architectures

for Binary Image Component Labeling

by

Quoc Tuan Pham

Master of Science in Computer Science

University of California, Los Angeles, 1987

Professor Sheila Greibach, Chair

We consider parallel algorithms in the PRAM (Parallel Random Access Machine) and SIMD (Single Instruction, Multiple Data Stream) models, and SIMD architectures for binary image component labeling, contour filling, component shrinking and component counting.

Current proposed SIMD architectures do not support efficient algorithms for the above three global operations (for images consisting of N pixels, the best time complexities of algorithms for component labeling are $O(N^{1/2})$ on the mesh-connected computer and $O(N^{1/4})$ on the pyramid computer). Current SIMD algorithms are complex, difficult to understand and highly dependent on the topology of the SIMD interconnection network.

We first give two O(logN) time, O(N) processor PRAM algorithms for the above operations. We then propose a general technique to map a certain type of PRAM algorithms into two proposed SIMD architectures with $O(\log^2 N)$ time penalty for PRAM concurrent read and concurrent write operations. In particular, we apply this technique to the above PRAM algorithms to obtain two $O(\log^3 N)$ time, O(N) processor SIMD algorithms which are more efficient than any other known SIMD algorithms for the above three operations.

1. INTRODUCTION

We consider 2 dimensional binary images (i.e. images whose pixels take on values 0 or 1) of size $N = n \times n$, $n = 2^k$ where k is an integer. Formally a nxn binary image B is a mapping from the array $\{0,\ldots,n-1\}^2$ into the set {0, 1} of pixel values. A pixel is defined to be a tuple (i,j) where $0 \leq i,j < n$. The four horizontal and vertical neighbors of a pixel are called its 4-neighbors. These and the four diagonal neighbors are called the 8-neighbors of the pixel. Figure 1 shows these two types of neighborhoods (or adjacencies) of a pixel. To avoid certain anomalies, we use different neighborhood types for the pixels with value 1 (the 1-pixels) and the pixels with value 0 (0-pixels) [KR]. From now on we just pick one type of neighborhood for the 1-pixels and keep it implicit in our discussion. We define a 3x3 neighborhood of a pixel to be the portion of the image consisting of the pixel and its 8-neighbors.

Consider a set S of pixels. The neighborhood relation on the pixels of S induces a graph, called the induced graph of S. In this graph, the vertices correspond to the S-pixels and the edges corespond to connections between adjacent S-pixels. We can define concepts analogous to those in graph theory such as paths and connected components. A digital path in S is a sequence of adjacent S-pixels. A set of S-pixels is said to be S-connected if every two pixels of the set can be connected by a digital path in S. A S-connected component (or S-component for short) is a maximal

connected set of S-pixels. Usually the set S is taken to be the set of 1-pixels. In this case we call the induced graph of S the induced graph of the image and drop the "S-" prefix from the terms defined above. We call the set of 1-pixels that have some 8-neighboring 0-pixels the boundary of a binary image.

In the component labeling operation, labels are assigned to the 1-pixels such that all pixels in every component have the same label and pixels from different components have distinct labels. In the contour filling operation, the pixels interior to the given boundary chains (defined in Section 3.2) representing the boundary of the binary image are "filled" with the value 1. In the component shrinking operation every connected component is shrunk to a distinct point. We must perform these basic operations as fast as possible since they may be called repeatedly.

Let f and g be functions from the natural numbers into the natural numbers. We say $f(n)=\Omega(g(n))$ ($O(g(n))$ resp.) if there exist a constant K and an integer n_0 such that for all $n>n_0$, $f(n)>= Kg(n)$ ($f(n)<= Kg(n)$, resp.).

We shall define the PRAM and SIMD models of parallel computers in the next Section. For sequential processing at least $\Omega(N)$ time is required to label the connected components of a binary images with N=nxn pixels [KR] since we must look at every pixel of the image. We can label the components of a binary image by applying

a component labeling algorithm for graph on its induced graph. There are many such O(logN) time PRAM algorithms (N is the number of vertices in the graph or the number of pixels in the image). However, the PRAM model is more difficult to realize than the SIMD model because of memory contention arising from the concurrent access by many processors to the same memory cell. The SIMD model is regarded as more realistic in image processing. Nassimi and Sahni [NS2] give an $O(N^{1/2})$ time SIMD algorithm for component labeling on the mesh-connected computer (MCC) using a tree data structure and two operations called random access read and random access write on the MCC. Miller and Stout [MS3] give an $O(N^{1/4})$ time SIMD algorithm for the pyramid computer (PC) using a similar data structure and data movement operations between pyramidal levels. Recently, Kumar et al.[KE] give an $O(\log^4 N)$ time SIMD algorithm on the mesh of trees computer.

In Section 3, two O(logN) time, O(N) processor PRAM algorithms for the above operations on binary images are derived from two different approaches __ tree merging as in [SV] and label propagation, a new parallel version of the raster scanning method given in [KR]. Nassimi and Sahni [NS1] showed that on the cube-connected computer (CCC) and the perfect shuffle computer (PSC), two SIMD architectures, we can simulate the CR and CW operations of the PRAM model in $O(\log^2 N)$ time for N processors. In Section 4, we propose two new SIMD architectures based on the CCC and the PSC. On these architectures, we can systematically adapt certain PRAM

algorithms, including those developed in Section 3, to corresponding SIMD algorithms with $O(\log^2 N)$ time penalty. Furthermore, these new SIMD architectures can simulate existing ones (the MCC, the PC and the MOT) with small penalties in time.

2. MODELS OF PARALLEL COMPUTATION

We now define the models for parallel computation on binary images of size (or the number of pixels) N=nxn, where $n=2^k$ for some k. They are the PRAM (parallel random access machine) model and the SIMD (single instruction multiple data stream) model. The latter includes the two dimensional mesh-connected computer (MCC) [NS1], the pyramid computer (PC) [Ta1], the mesh of trees computer (MTC) [KE], the cube-connected computer (CCC) and the perfect shuffle computer (PSC) [NS1].

2.1. **PRAM** (Parallel Random Access Machine)

A PRAM consists of a number of identical processing elements (PE) or processors, each with a unique id number, and a shared random access memory consisting of similar O(logN) bit cells where N is the image size. Figure 2 is a diagram of a PRAM. The PE's execute synchronously the same program in parallel. In each synchronous step a PE can perform either a basic computation, a concurrent-read (CR) access or possibly a concurrent-write (CW) access to an arbitrary cell of the shared memory. In the CREW (concurrent-read, exclusive-write) PRAM model, the CW operation can be performed only if different processors write to different memory locations (otherwise an error occurs and the machine halts). In the CRCW (concurrent-read, concurrent-write) PRAM, if many processors try to write to the same memory cell, there are different write-conflict

resolution policies so that only one value is chosen and written. Some of the many possibilities are: the mimimum value, the value of the processor with the minimum index, the common value (all processors must write the same value) and an arbitrary value chosen from the write values. The time (processor, resp.) complexity of a PRAM algorithm is the number of synchronous steps (processors, resp.) required for the computation as a function of the input size.

2.2. SIMD (Single Intruction-Multiple Data Stream)

A SIMD computer consists of a controller processor and a number of processing elements (PE's) with assigned distinct ids connected together by a fixed interconnection network. Figure 3 is a diagram of a SIMD computer. The processing elements are simple processors each with local memory consisting of a constant number of O(logN) bit cells where N is the number of pixels in the image. In each synchronous step, every PE satisfying a certain specified O(1) time simple test on its local parameters and variables executes the same instruction issued by the controller. The instructions each PE can perform include the normal basic arithmetic and logical instructions and the sending or receiving of an O(logN) bit long message through one of its links in the interconnection network. We can generalize this model by assuming that the controller can read the local memory of certain PE's. This capability is used for early termination detection of algorithms. The time (processor) complexity of a SIMD algorithm is the number of synchronous steps (processors)

required for the computation as a function of the input size.

We classify the SIMD computers by the topology of the interconnection networks. The followings are some existing interconnections.

2.2.1. MCC (Mesh-Connected Computer) [NS1, NS2]

A nxn MCC is a SIMD computer with nxn processors whose ids are (i,j), 0<=i,j<n. The PE with id (i,j) is connected to the processors whose ids are ($i \pm 1$, $j \pm 1$) (a connection exists only if the corresponding id exists).

For image processing applications, the original image is associated with at least a mesh-connected computer such that each pixel (i,j) is associated with a PE with the id (i,j). Hence each PE is connected to the PE's of the 4 neighboring pixels (see Figure 4).

An alternative method to assign ids to the PE's in the MCC is the shuffled row-major indexing scheme. Let $(x)_2$ be the number x written in binary representation (ranging from 0 to N-1). The shuffled row-major index of the PE (i,j) is the number whose binary representation is

$$(i_{k-1}\ j_{k-1}\ i_{k-2}\ j_{k-2}\ \cdots\ i_1\ j_1\ i_0\ j_0)_2$$

where $(i_{k-1}\ i_{k-2}\ \cdots\ i_1\ i_0)_2$ is the binary representation of i and

$(j_{k-1}\ j_{k-2}\ \ldots j_1 j_0)_2$ of j. The id of each PE is obtained by interlacing the binary representations of its coordinates. Thus the ids range from 0 to N-1. Figure 5 shows the shuffled row-major indexing of a 4x4 MCC. In Section 4, we shall present yet another scheme, the block indexing scheme, which is used to embed the MCC in the CCC.

2.2.2. PC (Pyramid Computer)[AS, Bu, Ta1]

The pyramid computer consists of k MCC's of sizes 2^k x 2^k , 2^{k-1} x 2^{k-1}, ..., 2^1 x 2^1 and 1 stacked up like a pyramid (see Figure 6a). The bottom nxn MCC contains the binary image. The PE which is the (i,j) PE of the hth MCC (of size 2^h x 2^h) where 0<=h<k has as its id the tuple (i,j,h) (see Figure 6b). A PE in level h is connected to its 4 son PE's on level h+1 and its parent processor on level h-1 as shown in Figure 6.

2.2.3. MOT (Mesh of Trees)[KE]

The MOT computer consists of a base level nxn MCC containing the binary picture and a full binary tree interconnection for each row and each column of the bottom MCC (see Figure 7).

2.2.4. CCC (Cube-Connected Computer)[NS1]

The processors have ids ranging from 0 to N-1. The processor with the id $(i_{2k-1}\ i_{2k-2}\ i_{k-3}\ \ldots\ i_1\ i_0)_2$ is connected to the 2k

processors with the ids

$$(i_{2k-1}\ i_{2k-2}\ i_{k-3}\ \ldots \underline{i_j} \ldots\ i_1\ i_0)_2$$

where $0 <= j <= 2k-1$, $\underline{i_j}$ is the negation of the bit i_j. This interconnection is the same as that of the hypercube computer (see Figure 8).

2.2.5. PSC (Perfect Shuffle Computer) [NS1]

In the perfect shuffle computer, the PE with the id

$$(i_{2k-1}\ i_{2k-2}\ i_{k-3}\ \ldots\ i_1\ i_0)_2$$

is connected to the processors whose ids are

$(i_{2k-1}\ i_{2k-2}\ i_{k-3}\ \ldots\ i_1\ \underline{i_0})_2$ (via the <u>exchange</u> link)

$(i_{2k-2}\ i_{k-3}\ \ldots\ i_1\ i_0\ i_{2k-1})_2$ (via the <u>shuffle</u> link)

$(i_0\ i_{2k-1}\ i_{2k-2}\ i_{k-3}\ \ldots\ i_1)_2$ (via the <u>unshuffle</u> link)

(see Figure 9).

The exchange links connect pairs of processors i and i+1 where i is even. The shuffle connection is similar to a perfect shuffle of a deck of N cards. The cards in the first half of the deck are intermixed with those in the second half. More precisely, the shuffle link connects the processor i to the processor

$$S(i) = \begin{cases} (2i) & \text{if } 0<=i<=N/2-1, \\ (2i+1-N) & \text{if } N/1<=i<=N-1 \end{cases}$$ [St].

2.2.6. 2^b-Blocks

Assume the ids of the PE's of a SIMD computers are numbered from 0 to N-1 with the assumption on N as before. We define a subset of 2^b PE's, where $0<=b<2k$, whose ids have the same first 2k-b bits a 2^b-block. If $0<=b<2k-1$ then a 2^{b+1}-block consists of two 2^b-blocks where the bth bits of the binary representations of the ids of the PE's in the first block (called the left block) all equal 0 whereas those of the second block (called the right block) 1. Figure 10 shows a MCC with the shuffled row-major indexing and its 2^b-blocks. The divide and conquer strategy can be implemented on the SIMD model to compute the solution to a problem as follows. In each phase, we compute in parallel the solutions for the 2^{b-1}-blocks and then merge the solutions of pairs of corresponding left and the right 2^{b-1}-blocks into solutions of the 2^b-blocks. An example of this strategy can be found in Section 4.3.

3. ALGORITHMS ON PRAM's

The component labelling operation for binary images is an instance of the general component labeling problem for graphs where the degrees of the vertices are bounded by 4- (8-, resp.) for 4- (8-, resp.) connectedness. With this view, in Section 3.1 we modify a component labeling algorithm for graphs by Shiloach and Vishkin [SV] to a O(logN) time, O(N) processor component labeling algorithm for binary images of size N. In Section 3.2, we exploit the special topological structure of the connected components of binary images to develop a new O(logN) time, O(N) processor algorithm using a strategy similar to the sequential raster scanning of horizontal rows of a binary picture.

3.1. The Tree-Merging Algorithm

In [HCS], Hirschberg et al. develop a parallel component labeling algorithm for graphs on the CREW PRAM model with $O(\log^2 V)$ time, $O(V^2)$ processor complexity where V is the number of vertices. Improved algorithms based on variations of their strategy include those of Nassimi and Sahni [NS2] for graphs of bounded degrees ($O(\log^2 N)$ time, O(V) processors using the min value CRCW PRAM), Shiloach and Vishkin [SV] (O(logV) time, O(V+2E) processors using CRCW PRAM with arbitrary write conflict resolution) and Gazit [Ga] (random O(logV) time, O((V+E)/logV) processors using the CRCW PRAM).

3.1.1. Shiloach and Vishkin's Algorithm [SV]

The algorithm uses the CRCW PRAM model where concurrent-write is arbitrary, i.e. only one of the write values is arbitrary chosen and written. We first need a number of definitions:

Basic Definitions

1. A rooted tree (or tree, for short) is a directed graph satisfying:

   a. There is a unique self-loop at a vertex called the root;

   b. The underlying graph minus the self-loop is a tree; and

   c. There is a directed path from each vertex to the root.

   The height of a rooted tree is length of the longest simple path from a leaf to the root.

   A rooted star is a rooted tree such that each vertex is connected directly to the root, i.e. a rooted tree with height at most one.

Notations

a. In the formal definition of a PRAM, the processors can only access the memory cells via their addresses. For convenience, we can assume that in our parallel programming language array indexing of memory locations is possible (as in PASCAL) since computation of the address offset from a base address only takes O(1) time. Hence it is possible to declare a global array variable A(0..(N-1)) and refer to its ith element by A(i). We can also have tuples of integers as indices.

b. We use "<-" for variable assignment. The statement "A(v) <- u" means "assign the value u to the vth element of the array A". The statement "A(v) <- A(A(v))" means "let r be the value stored at A(v), assign the value stored at A(r) to the variable A(v)".

Let G =(V,E) be the graph whose components are to be labeled. We assume that the vertex set V is {0,...,N-1}. We declare the global pointer variable D(0..(N-1)). The variable D represents a conceptual graph (changing over time) with vertices in V and and edges (v,D(v)) from each node v to node D(v).

The goal of the algorithm is to construct a pointer graph D (on top of the original graph) in the form of a forest of rooted stars each of which contains exactly all the vertices of a connected component of G. Then the labels of the vertices can be obtained in constant time from these rooted stars.

We can view the connected components as a partition P of the set V. The algorithm first puts each vertex in a rooted D-tree, i.e. we start out with the finest partition of V: the processor associated with each node v set its D(v) variable to v. In each iteration of the algorithm, the rooted D-trees are repeatedly merged with their "adjacent" rooted D-tree(s) (rooted trees representing other subsets of the same components) such that the new induced partition is coarser but remains at least as fine as than the component partition P. This tree merging operation is called tree hooking which can be

defined formally as follows.

2. Let T and T' be two distinct rooted trees in the pointer graph D, r the root of T and v a vertex in T'. The operation "D(r) <- v" is called the hooking of T into T'. It connects the (old) root r of T to the node v in T'.

To guarantee that only a logarithmic number of steps are required to obtain the final partition P in the form of a forest of rooted stars we need to intermix the mergings of rooted trees and the pointer skipping operations defined as follows.

3. The pointer skipping operation on the pointer graph induced by the variable D(v)'s is the operation:

For all v in V **do in parallel** D(v) <- D(D(v)).

Before this operation, each node v points to node D(v). After the operation, each node v points to the node to which the node D(v) used to point.

We assume that the assignment is carried out only if the right hand side is not nil. If the pointer graph is a linear linked list (rooted tree, resp.) of length (height, resp.) L then after $[\log_{3/2}L]$ jumps ([x] denotes the ceiling of x), every vertex (except the last one) points to the last node (the root, resp.). Figure 11 illustrates the pointer skipping operation on a list of size 8.

An Informal Description of Shiloach and Vishkin's Algorithm [SV]

Initialization:

For all v do in parallel D(v) <- v;

The algorithm performs the following iteration at most $\{\log_{3/2}N\}+2$ times ($\{x\}$ denotes the floor of x):

Iteration s:

(Let D_s denote the value of D after the sth step)

1. Perform a pointer skipping on D.

2. Hooking trees into smaller vertices of other trees: Each vertex i that pointed to the root r at the end of the previous iteration (hence whose pointer D(i) unchanged in step 1) checks whether it has a neighbor j (in the original graph) pointing to a vertex with a smaller id. If so it tries to hook its tree onto that neighbor's tree by setting the root's D-value to the neighbor's id:

if $D_{s-1}(i) = D_s(i)$ (= r) **then**

for every edge (i,j) of G **do**

if $D_s(j) < r$ **then** $D_s(r) \leftarrow D_s(j)$;

Here two processors are required for every edge in the graph G (one for each direction) and the processors associated with

the edges perform arbitrary CW to resolve the case where many edges try to write to the same D(r) variable.

3. Hooking stagnant trees: A tree is stagnant in the sth iteration if it has not been changed in the first two steps of this iteration (i.e. it is already a star so there is no change by the skipping operation or it was not hooked onto another tree and no other tree was hooked onto it) and its root is called a stagnant root (it is possible for a root to know whether it is stagnant if in steps 1 and 2 every processor that changes its pointer writes an appropriate value to a special variable of the new node it now points to). Every vertex pointing to a stagnant root checks whether it has a neighbor j pointing to a vertex of another tree. If so, it tries to hook its tree onto the node j points to. Here the processors associated with the edges in the original graph (i,j) do the necessary computation to find out stagnant roots and to hook stagnant trees.

4. Perform another pointer skipping on D.

Shiloach and Vishkin [SV] prove the correctness and the logarithmic time complexity of the algorithm by a series of 12 lemmas. The following is a sketch of their proof.

a- The pointer graph obtained at any point during the computation is a forest of rooted trees since the followings hold.

1. If there is a simple path between two vertices then it is unique since the outdegree of every node is exactly 1.

2. A leaf (a vertex with indegree 0) never obtains a child because of the hooking rules in step 2 and 3.

3. Two vertices pointing to two different stagnant roots after step 2 are not adjacent in the original graph, or else one of the roots would have been hooked to the other one.

4. If a rooted tree is stagnant after step 2 of an iteration then its root is at height at most 1 after steps 2 and 3 because it is already a star (there is no change after step 1) and in steps 2 and 3 no other tree can be hooked onto any of its nodes.

5. If a vertex points to another vertex with larger label after steps 1, 2 or 4 then it is a leaf in the pointer graph after steps 1, 2 and 4 respectively since a node acquires a larger father only in step 3 and then the pointer jump in step 4 turns it into a leaf.

b- If a root does not change during an entire iteration then its tree contains all the vertices of a connected component and so remains stagnant to the end of the computation.

c- If a rooted tree changes in the sth iteration (if a tree hooks onto another tree then it ceases to be a rooted tree on its own) then the number of nodes in the tree is at most $(3/2)^{s-1}$ times the height of the tree. Hence the rooted trees are merged together at a rate that guarantees termination after $\{\log_{3/2}N\}+2$ steps.

3.1.2. Modifying the Tree-Merging Algorithm for Binary Image Component Labeling

The component labeling operation on a binary image can be reduced to the component labeling problem on its induced graph. Shiloach and Vishkin's component labeling algorithm requires a processor for each vertex and two processors for each edge of the graph. Since the vertices of the induced graph have bounded degrees (for the 0-pixels: zero, for the 1-pixels: at most 4 in 4-neighborhood adjacency and at most 8 in 8-neighborhood adjacency), we need only one processor per vertex (pixel). This "vertex" processor can do the works of the processors associated with the outgoing edges from the vertex since all the "edge" processors perform similar tasks in each step. Hence we only need O(V) processors instead of O(V+2E) processors in the general Shiloach and Vishkin's algorithm. The complexity term only increases by a constant factor. Algorithm 1 in Appendix 1 is a detailed description of the modified component labeling algorithm for the binary images. Figure 12 shows how Algorithm 1 works on a binary image.

3.2. THE LABEL PROPAGATING ALGORITHM

In this section, we derive a new parallel version of the component labeling algorithm similar to the method of propagating labels from boundaries of connected components and raster scanning as suggested in [KR]. In Section 3.2.1 we introduce the necessary facts in digital topology. Then we present the Label Propagating Algorithm in Section 3.2.2.

3.2.1. Further Definitions in Digital Topology

In Section 1, we introduce the concepts of neighborhood types. Because of certain anomalies (see [KR]), we have to use different neighborhood types for the 1-pixels and the set of 0-pixels. Consider the set of 0-pixels and its components (with respect to 4-(8-, resp.) neighborhood if we use 8-(4-) neighborhood for the 1-pixels). We define the background of the binary image to be the union of the 0-components whose 0-pixels are connected to the border of the image. Other 0-components (if exist) are called the holes of the image. The boundary of a binary image is the set of 1-pixels adjacent to some 0-pixels. The boundary of a (connected) component is defined likewise. It is the union of two types of sets __ the outside boundary and the inside boundary. The outside (inside, resp.) boundary consists of 1-pixels adjacent to the 0-pixels of the background (a hole, resp.). A connected component has one outside boundary and zero or more inside boundaries.

Consider the pattern of pixels

1 1

1 1

in 4-neighborhood adjacency and the patterns of pixels

1 0 0 1 1 1 1 1

1 1 1 1 1 0 0 1

in 8-neighborhood adjacency. They induce subsets of R^2 (in which we embed the image grid) in the forms of a square and triangles, respectively. The above patterns of pixels in each connected component C of the image induce a subset (or region) P(C) of R^2 in the natural way. As in plane set topology, P(C) has an outside boundary and some inside boundaries each surrounding one of P(C)'s holes. The outside (an inside, resp.) boundary of P(C) contains the some outside (inside, resp.) boundary pixels. It is well-known that the boundaries of P(C) can be represented by a collection of oriented curves so that the outside (each inside, resp.) boundary is a clockwise (counterclockwise, resp.) oriented curve. These oriented boundary curves induce a directed graph B(C), called the boundary chains of C, on the set of boundary pixels. B(C) consists of a clockwise oriented cycle called the <u>outside boundary chain</u> of C and a number of counterclockwise cycles called the <u>inside boundary chains</u> of C. The outside boundary chain contains some outside boundary pixels and surrounds all the pixels of C. Each inside boundary chain contains some inside boundary pixels and surrounds a hole of C. (Our method of representing the boundary of a component by a pointer

structure is similar to Freeman's chain codes [Fr].) Figure 18 shows a component and its boundary chains.

We call a (boundary) pixel singular if it appears more than once on the outside boundary chain or if it appears on more than one boundary chain. It is possible, to compute the outgoing and incoming links of the boundary chain at a boundary pixel by only looking at it 3x3 neighborhoods. Appendix 2 shows how this can be done. Since there is a finite number of possible 3x3 neighborhoods, it is obvious that the time complexity to compute the boundary chains is O(1).

There are many technical points concerning the structure of the boundary chains in the most general case. We first present the label propagation algorithm for simple cases and then show the necessary modifications for the general case.

3.2.2. Description of the Label Propagating Algorithm

Imagine a component in a binary image to be like a continent. We can think of its holes as lakes and its outside boundary as the coastline. It is possible that there are other components inside a hole (just as islands in a lake). Each of these nested components, in turns, has its own coastline and lakes.

For each component, we first propagate the minimum coordinates of the pixels on each boundary chains along the whole chain. Then we

propagate horizontally from left to right the minimum label of the outside boundary chain to other pixels in the component. In our analogy, a continent is systematically explored by first exploring along the coastline and all the lake shores and then the land mass from West to East.

First we consider the simplest case where there is no thin line in the image and no two boundary chains touch each other (i.e. there is no singular pixels). In this case, every boundary pixel has in-degree and out-degree equal to 1 in the induced graph of the boundary and the maximum length of a boundary chain is N. After O(logN) steps, each consisting of a pointer skipping and a propagation of the larger coordinates along the pointers, every node on each boundary chain receives the same label (namely the lexicographically minimum of the coordinates of the pixels on the chain) and because of the disjointness of the boundary chains, nodes from different boundary chains receive different labels (see Figure 13).

If it is known *a priori* that there is no hole in the induced graph, i.e. the image consists of solid figures with no thin line, then in O(1) time we can set up the linked lists of the form 11..11 of length at most n as follows. Every 1-pixel sets a pointer to its East-neighboring 1-pixel (if exists). Hence each of these lists represents a run of consecutive 1-pixels from a left (outside) boundary pixel to the corresponding next right (outside) boundary

pixel of a component. After O(logn) pointer skippings, every pixel in each of such lists has its pointer set to the last element of the list on an outside boundary and then reads and uses the last element's label as its component label.

Now suppose that there are holes, hence inside boundary chains. We would like the pixels on the boundary chain to find out whether they are on an inside or an outside boundary chain. The strategy is to use the coordinates (0,0) as the reference point and one elected node on each chain to find out the orientation of the whole chain. Every minimum pixel on each boundary chain can determine whether it lies on an outside (the local orientation is counterclockwise) or an inside (clockwise, resp.) boundary chain by looking at its incoming and outgoing links in the boundary chain [WBR]. Every boundary pixel can find out whether it lies on an inside or an outside boundary chain by consulting the corresponding minimum pixel on the chain. Hence in a constant number of steps (after the computation of the mimima of the chains), every pixel can find out whether it is on an inside or outside boundary chain of a component. If so, it can further find out whether it is on the left (West) side or on the right (East) side (or neither) of the boundary chain by looking at its East and West neighbors.

If it is known *a priori* that there is no 1-pixel inside any hole, i.e. no island in any lakes, then we can set up in O(1) time the linked lists representing horizontal runs of pixels of the form

10..01 of length at most n, where the first 1 is on a left inside boundary pixel and the second on the corresponding right inside boundary pixel on the other side of the lake. Apply the pointer skipping operation O(logn) times, the beginning 1's can find out the coordinates of the corresponding ending 1's, i.e. the horizontal bridges over the lakes have been constructed, and the rest is the same as before.

If there are components nested within components (see for example Figure 14), a more novel approach is required to set up the correct lists of horizontal runs of 1-pixels. We want to set up linked lists the form 1...1*1...1*1...1*1...1 where each "1..1" corresponds to a horizontal run of consecutive 1-pixels of a connected component beginning at a left outside (right inside, resp.) boundary pixel and ending at the next corresponding left inside (left inside or right outside boundary pixel, resp.), and each star denotes a bridge over the across a hole (and over the islands in the hole) of the component from a left inside boundary pixel to the corresponding right inside boundary pixel on the other side of the hole (see Figure 14). After O(logn) pointer skippings every node in each of the list will point to the last node on a right outside boundary position and can read the correct component label there. Figure 15 illustrates how the algorithm works.

It remains to show how to set up the stars efficiently in O(logN) time. We associate a left (right, resp.) parenthesis to each

of the left (right, resp.) inside bank pixel and a '#' to other pixels. Therefore assigned to each row of the binary image is a string in the alphabet {(,),#} such that the left and right parentheses are balanced (see Figure 14). The problem of matching the corresponding left and right banks (i.e. setting up the stars) is then reducible to the problem of matching pairs of parentheses.

3.2.2.1. An Algorithm to Match Pairs of Parentheses

Bar-On and Vishkin [BV] give an algorithm using a binary tree structure to solve the following problem:

Matching Pairs of Parentheses:

Given a string w of balanced parentheses of length |w|= n stored in memory locations 0... n-1 (one character at each location). Compute in parallel, for each left parenthesis the position of its matching right parenthesis.

Without loss of generality, we can assume that n is a power of 2. The algorithm requires 2n-1 processors and uses a data structure in the form of a complete binary tree with height [logn] and the n symbols of the string w stored from left to right at the leaves. We allocate a (leaf) processor to each parenthesis and (n-1) processors to the internal nodes of the binary tree. Figure 16 illustrates the binary tree and the assignment of processors for a string of length 16.

First, the levels of nesting of the parentheses can be computed as follows. (Let L(0..n) be a array variable.)

1- For each leaf i, assign to L(i) the value +1 (-1, resp.) if location i contains a left (right, resp.) parenthesis; and

2- Compute the prefix sum L(i), 0 <= i < n, on the values v, i.e. for the parenthesis at location i, we compute the sum of all the numbers v(j) where 0<= j <= i. Basically, we can do so using the above binary tree structure in two phases each in O(logn) steps. In the first phase, the values are passed up in parallel, level by level, from the leaf level to the root level. The values received each internal node are added together and the sum is sent upwards. In the second phase, the values are passed down in parallel, level by level, from the root level to the leaf level. Each internal node adds **the values it received from its parent and its left child and then** send the sum (zero, resp.) down to the right (left, resp.) child.

3- If a leaf i contains a right parenthesis then it adds 1 to the variable L(i).

Now L(i) is the level of the nesting of the ith parenthesis.

Example: For the string ((( ) (( ))) ( )), the level of nesting computed as above is

L: (1,2,3,3,3,4,4,3,2,2,2,1).

4- Note that the position of the matching right (left, resp.) for the left (right, resp.) parenthesis at position i is the smallest (largest, resp.) j such that j>i (j<i, resp.) and L(i)= L(j). We then compute in O(logn) time for each internal node j the two values MINLEFT(j) and MINRIGHT(j) where

MINLEFT(j) = min{L(i)| i is a leaf node of the subtree rooted at j and position i contains a "("}

MINRIGHT(j) = min{L(i)| i is a leaf node of the subtree rooted at j and position i contains a ")"}.

The computations of the MINLEFT and MINRIGHT values are similar to the computation of the prefix sum. However only the first phase with minimum computation instead of summation at each internal node is necessary.

Each leaf processor then uses the above values to search in parallel with other leaf processors for the position of the matching parenthesis. Consider the left parenthesis at location i looking for its matching right parenthesis at location j. (Note that all leaves between i and j have L-values greater than L(i).) It is searching for the first leaf j to its right with L(i)=L(j) by performing the following actions:

- Find the least common ancestor ancestor k of i and j by climbing up the tree from the leaf i to the first node whose right child k satisfies MINRIGHT(k) <= L(i); k must be an ancestor of j.

- Move down the tree from node k to j, always to the left if possible: if we are at a node with left child g and right child h move down to g if MINRIGHT(g) <= L(i), otherwise move down to h.

The time complexity of this search is O(logn) since each leaf processor moves up the tree at most once and down at most once.

The above algorithm can be adapted to our purpose as follows.

- It is easy to generalize the algorithm to string of balanced parentheses on {(,),#} where the # characters of the string are ignored.

- Instead of using 2n-1 processors for strings of size n, we can use only n processors by assigning to each processor a leaf and at most a distinct internal node. The complexity of the algorithm increases only by a constant factor. Figure 17 shows how this can be done for n = 8.

- Each row of the binary image is associated with a separate binary tree. The matching of pairs of parentheses is done in parallel for all the rows.

3.2.2.2. A Detailed Description of the Label Propagating Algorithm

We assign a processor PE(u,v) for each pixel (u,v). Algorithm 2 in Appendix 3 is a detailed description of the Label Propagating Algorithm for the simple case.

3.2.2.3. Modifications for the General Case

Here we show how to modify the Label Propagating Algorithm for the general case where there exists a singular pixel.

A pixel can be on a boundary chain up to 8 times (see Figure 19). Hence the length of a boundary chain is at most 8N. Furthermore a pixel can appear on as many as 4 different boundary chains. It is easy to modify Algorithm 2 to accommodate these two types of possible repeated occurrences (each PE splits up to 4 conceptual PE's and has to simulate as many processors as required by its repeated occurrences). Using pointer skipping and the propagation of the larger coordinate along the links, in

$$O(\log(8N)) = O(\log(2^{3+2k})) = O(2k+3) = O(\log N+3)$$

steps, we can make sure that all the nodes on each boundary chain get the same label and nodes from different boundary chains have different labels. We can also incorporate early termination detection to find out when further pointer skipping will not change the pointer structure.

The determination of the orientations at the minimum points of the boundary chains can be also modified for the general case. The only complication arises when a boundary chain passes through the minimum point more than once and the rule for determining the orientation gives ambiguous answers (for example, one local segment gives the answer "counterclockwise" and another "clockwise"). Again

in this case the problem is resolved by conceptually splitting up the singular node and letting it acts appropriately and independently for each of these chains (See Figure 20).

The time complexity of this component labeling algorithm is O(logN) and the processor complexity is O(N).

3.2.3. Contour Filling, Component Shrinking and Component Counting Algorithms

The Label Propagating Algorithm can be modified easily into other algorithms for related operations with similar complexity. Recall that in the contour filling operation, given a collection of boundary chains representing the components of a binary image, we would like to recover the original binary image. We just set up linked lists of horizontal runs of pixels of the form 10...01, where the first (last, resp.) 1 is a left outside or an right inside (inside left or right outside, resp.) boundary pixel, and propagate the 1's along the elements of these lists.

A parallel component shrinking algorithm can be obtained by applying a component labeling algorithm and then setting the values of all 1-pixels with component labels not equal to their own coordinates to 0. Using a tree like computation we can count how many 1-pixels remain after this conversion hence obtain an algorithm for component counting.

4. MAPPING PRAM ALGORITHMS INTO SIMD ARCHITECTURES

The abstract CR and CW operations in O(1) time facilitate the design and analysis of parallel algorithms on the PRAM model. However in practice, it is easier to build the interconnection network of a SIMD machine than the shared memory of PRAM. On the other hand, it is quite difficult to come up with SIMD algorithms in an *ad hoc* way for the particular topology of the interconnection network. In Section 4.1, we give a method to map the two PRAM algorithms developed in Section 3 into SIMD architectures by simulating the CR and CW operations by the RAR and RAW operations on the SIMD model. In Section 4.2, we give a brief survey of existing SIMD algorithms for component labeling. In Section 4.3, we present Nassimi and Sahni's algorithms for the RAR and RAW operations on the CCC and the PSC. Finally in 4.4, we propose two SIMD architectures based on the CCC and the PSC. On these two architectures we obtain SIMD component labeling algorithm with current best time complexity using the simulation method in Section 4.1.

4.1. Simulation of a Class of PRAM Algorithms on the SIMD Model

Suppose that in a PRAM algorithm, the following conditions are satisfied:

1- For an input of size N, the algorithm requires N processors and a finite number of array variables A_1, A_2, ..., A_c, where c is a constant independent of N, and the array indices are in the set {0,...,N-1}; and

2- In each synchronous step, each processor PE(i) can only perform the CR or CW operation on the same array variable $A_j(f(L(i)))$, where j is in {1,...,c}, L(i) is PE(i)'s local memory state and f is a function computable (at each processor) in constant time.

(The two algorithms in Section 3 satify the above two conditions and these conditions define a reasonably large class of PRAM algorithms.) Then on a SIMD computer with N processors with ids {0,...,N-1}, we can run the PRAM algorithm as follows. For each i, 1<=i<=c, we allocate in the local memory of each SIMD processor PE(i) space for the variables $A_1(i)$, $A_2(i)$ ..., $A_c(i)$. Note that we restrict the size of the local memory of every SIMD PE to a constant number of O(logN) bit locations. The SIMD PE's can write and read data to and from the local memory of each other by routing data through the interconnection network. The PRAM CR and CW operations on the shared

memory can be simulated on SIMD architectures by the two operations called Random Access Read (RAR) and Random Access Write (RAW) defined as follows. (As in the PRAM model, we assume that in the SIMD model it is possible to declare a global array variable A(0,...,N-1) and to refer to the local "component" of A at the ith processing element as A(i).)

Definitions [NS1]

1. RANDOM ACCESS READ (RAR(S, D)):

Let S and D are two declared array variables. S contains the read destinations. D contains the data to be read. Each processor PE(i) is to receive the value of the variable D(S(i)) held in the local memory of the processor PE(S(i)). If S(i) = ∞ then PE(i) is not to receive any value from any processor.

2. RANDOM ACCESS WRITE (RAW(S,D)):

Let S and D are two declared array variables. S contains the write destinations. D contains the data to be read. The processor PE(i) is to transmit the value of its local variable D(i) to the processor PE(S(i)). If S(i) = ∞ then PE(i) is not to write to any processor. Conflicting writes to the same PE may be resolved by picking the min write value or the write value of the PE with min index.

Since it is not feasible for large N to connect every pair of PE's directly, to perform the RAR and RAW operations we must route

data through the interconnection network. If the RAR (RAW, resp.) is exclusive (similar to ER and EW on the PRAM) and every processor want to read (write resp.) then it can be reduced to the sorting operation on record with the processor indices as keys. For example, to do exclusive RAW we set up the records [S(i), D(i)] at PE(i), 0<= i< N, and sort them on the field S(i).

If many PE's try to read or write to the same local variable of a PE and sorting alone will not work because of the conflicting reads or writes. Nassimi and Sahni [NS1] showed that RAR and RAW in the general case can be implemented using the following operations:

1.SORT(R,K): (Sorting R on its key field K)
Each processor PE(i) contains the record variable R(i) with key field K(i), 0<= i < N. Following the sort operation, the records R(i)'s will be rearranged in the processors such that the processor PE(j) contains the record R with the jth largest K-field.

2. RANK(T,r) : (Ranking on the tag T into r)
This operation is similar to the prefix sum. Each processor PE(i) has a tag variable T(i) and is selected if its tag equals 1. The operation rank assigns to the variable r(i) of each processor PE(i) the number of selected processors among the processors PE(0), ..., PE(i-1).

3. CONCENTRATE(R,r): (Concentrate the records R on the rank r)
Assume that the processors contain the variable r(i)'s containing the ranking on a tag variable T. A concentrate operation on the record R moves the record R(i) to the processor PE(r(i)).

4. DISTRIBUTE(R,d): (Distribute the records R on d)
Let R(i), d(i), be variables in the processor PE(i) such that $0 \leq d(i) < d(j) < N$ if $0 \leq i < j < N$. The distribute operation routes each record R(i) of PE(i) to the processor PE(d(i)).

5. GENERALIZE(R,d): (Generalize R on d)
This operation is similar to the distribute operation except that the processors PE(d(i-1)+1),...,PE(d(i)-1) also receive the record R(i) (we assume that d(-1)=0).

Nassimi and Sahni's method is best illustrated by means of an example. Given the above subalgorithms, The RAR(S,D) operation is carried out as follows [NS1].

- We start out with the following configuration:

i:	0	1	2	3	4	5	6	7
S:	2	6	2	∞	5	6	∞	6
D:	a	b	c	d	e	f	g	h .

- Each processing element PE(i) sets up the record

$$G(i) = [S(i), T(i)=i, F(i)=1]$$

where S(i) contains the id of the PE whose D-value is to be read. T(i) marks the origin of the record R(i) before the sorting in the next step.

- Sort(G,S): Sort the G-records on the first key with ties being resolved on the T-values. The value ∞ is considered to be the infinite value. During the sorting, if for some i and j, S(i)=S(j) and T(i)<T(j) then Flag(i) is set to 0 so that after the sorting only records with distinct values have nonzero flags. Each PE keeps the new S, T and F values of the G-records they received. The new configuration is as follows.

i:	0	1	2	3	4	5	6	7
T:	0	2	4	1	5	7	3	6
S:	2	2	5	6	6	6	∞	∞
F:	0	1	1	0	0	1	.	.
D:	a	b	c	d	e	f	g	h

- We then rank the processors with F-value of 1. The rank values will be stored in the variable r. The resulting configuration is as follows.

i:	0	1	2	3	4	5	6	7
T:	0	2	4	1	5	7	3	6
S:	2	2	5	6	6	6	∞	∞
F:	0	1	1	0	0	1	.	.
r:	.	0	1	.	.	2	.	.
D:	a	b	c	d	e	f	g	h

- For each PE(i) with F-value 1, we define a new record G' as follows.

G'(i) = (r(i), U(i)=i, S(i))

- CONCENTRATE(G',r). The new configuration is as follows.

i:	0	1	2	3	4	5	6	7
T:	0	2	4	1	5	7	3	6
S:	2	5	6	.	.	.	.	.
U:	1	2	5	.	.	.	.	.
D:	a	b	c	d	e	f	g	h

- Each PE(i) receiving a G'-record sets up the record

G"(i) = (S(i), V(i)= i).

- We then perform DISTRIBUTE(G",S) and obtain the following configuration:

i:	0	1	2	3	4	5	6	7
T:	0	2	4	1	5	7	3	6
S:	.	.	2	.	.	5	6	.
V:	.	.	0	.	.	1	2	.
D:	a	b	c	d	e	f	g	h

Note that a PE receives a G"-records iff it contains a value to be read by another PE. We now uses T, U, V to route these values to the reading PE's.

- Each PE(i) receiving a G"-record sets up the record

R(i) = (K(i)=D(i), V(i)).

Perform CONCENTRATE(R,V), i.e. concentrate the D-values on the return addresses V(i)'s.

i:	0	1	2	3	4	5	6	7
T:	0	2	4	1	5	7	3	6
V:	0	1	2	.	.	.	.	.
K:	c	f	g	.	.	.	.	.
U:	1	2	5	.	.	.	.	.
D:	a	b	c	d	e	f	g	h

- Perform GENERALIZE(K, U): generalize the read values using the return addresses U(i)'s.

i:	0	1	2	3	4	5	6	7
T:	0	2	4	1	5	7	3	6
K:	c	c	f	g	g	g	.	.
D:	a	b	c	d	e	f	g	h

- Perform SORT(K,T). We have routed the read values to the appropriate reading PE's.

i:	0	1	2	3	4	5	6	7
K:	c	g	c	.	f	g	.	g
D:	a	b	c	d	e	f	g	h

Similarly, RAW(S,D) can be carried out as follows.

- Each processor PE(i) sets up the record

R(i) = [S(i),T(i)=i,D(i),Flag(i)=1]

where S(i) is the destination id of the write and D(i) is the value to be written.

- Sort the above records on the S-field. Whenever a comparison is done for two records with the same S-value, the one with smaller T- (D-, resp.) value sets its flag to 0 if min-index (min-value, resp.) concurrent-write resolution is desired. (Other types of write-conflict resolutions can be achieved by modifying the sorting algorithm accordingly.)

- Rank the processors with nonzero flags and concentrate their R-records (obtained in the sort) the on the ranking values.

- Distribute on the S(i) keys and we have routed the write values to their specified destinations.

The time complexities of RAR and RAW are the time complexity of the most time consuming suboperation. In certain special cases of RAR and RAW, we do not have to carry out certain substeps.

Examples:

1- Distinct PE's read from (write to, resp.) to different processors and every PE reads from a PE. This is almost similar to ER (EW, resp.). Since the S vector is a permutation of (0 1 2 ... N-1) we only need to perform two (one, resp.) sorting operations to route the data to the correct destination. The first one is to reach the right destination of the read operation. The second is to bring back the obtained read value.

2. Termination detection: Every processor that detects a change in value of a variable in a phase of the computation changes its detection variable to 1 from the initial value 0. Then we do a suffix sum operation (similar to the computation of the nesting of level of parentheses in Section 3.2.2.1). In our extended SIMD model, we assume that the controller has access to the local memory of

PE(0). Hence after the suffix sum the controller can find out if there is any selected processor or any variable that was changed before the suffix sum operation.

The above algorithms for RAR and RAW provide us a uniform and powerful tool for data communication on the SIMD model to simulate CR and CW in the class of PRAM algorithms defined above. In the following sections we shall give a brief survey of existing SIMD architectures and algorithms for the operations of interest and propose two new architectures which support these operations with $O(\log^3 N)$ time, $O(N)$ processor algorithms.

4.2. A Survey of Existing Component Labeling Algorithms on SIMD Computers

Recall than N=nxn is the size of the image. The 2 dimensional MCC is very efficient for local operations such as boundary computations. The PC is further quite efficient for computing the minimum, the maximum, the average pixel values (for gray images), multiresolution operations [Ro3, Ta2] and other operations. These two SIMD models have motivations from natural biological vision systems [HR, Ta1, U]. However they are not efficient and are difficult to program for global operations such as component labeling, contour filling and component shrinking. The O(n) parallel MCC algorithm developed in [NS2] and the $O(n^{1/2})$ PC algorithm in [MS3] use the routing techniques similar to that of [NS1]. It is easy to see that a global operation in a MCC must take $\Omega(n)$. For the PC, a heuristic argument based on the movements of $O(n^2/2)$ data elements from the left half of the base picture MCC to the right half though a cut of O(nlogn) links requires about $\Omega(n/\log n)$ time. In fact there is a more precise formulation of a lower bound for the pyramid computer in [MS3]. Recently Kumar et al give an $O(\log^4 n)$ component labeling algorithm on the MOT computer. However in this architecture it is easy to see that the bottleneck at the roots of the binary trees prevents an efficient implementation of the pointer skipping operation used in the contour filling.

4.3. Efficient Data Movement Operations on the CCC and the PSC

On the CCC and PSC we can perform the sorting of N elements in $O(\log^2 N)$. The CCC (PSC, resp.) sorts by emulating the mergesort network of Batcher [Ba] (the perfect shuffle network of Stone [Kn,St], resp.).

Nassimi and Sahni [NS1] have shown that the rank, concentrate, distribute and generalize operations can all be done efficiently in O(logN) steps using the divide and conquer strategy on the 2^b-blocks.

Examples:

The following is a brief description of Nassimi and Sahni's algorithms [NS1] for the RANK, CONCENTRATE, DISTRIBUTE and GENERALIZE operations on the CCC.

1. The RANK Operation.

The rank of a selected record R in a block is the number of selected records in the same block which reside in the PE's with smaller indices. We compute the rank independently and in parallel for the 2^b-blocks. The block B consists of left (L) and right (R) 2^b-blocks (see preliminaries). The rank of S in B is:

- the rank of S in L if S is in L;

- the rank of S in R plus the number of selected records in L.

Converting this recursion into an iterative version, we have the following program for the rank operation on the CCC:

```
procedure Rank(SEL,H);
{procedure to rank the selected records in the N PE's}
{Initially, SEL(i)=1 if the record in PE(i) is selected;
 at the end, H(i) contains the rank of i if PE(i) is selected}
{PE(i) executes the following program:}
H(i) <- 0; S(i) <- SEL(i);
for j:= 0 to b-1 do
begin {compute the rank for the 2^(j+1) blocks}
        {i^(b) is the number obtained by negating
         the b^th bit of i}
        T(i^(j)) <- S(i); {done via a CCC link}
        if i_j, the j^th bit of i is 1 then
           {update the rank in the right blocks}
           H(i) <- H(i)+T(i);
        S(i) <- S(i) + T(i);
        {get the number of selected records
         in the 2^(j+1) blocks}
end
if SEL(i)=0 then PH(i) <- undefined;
```

2. The CONCENTRATE Operation

The procedure CONCENTRATE(G,r) moves the record G(i) from PE(i) to the r(i)th PE. r contains the rankings (in the block) as determined by the rank operation. Concentration is performed in logN phases. In the pth phase, the G-records are moved within the 2^p-blocks so that each PE's index and the rank of the G-record residing in it agree in bits 0,...,p. We make r(i) a field of G(i) so it is routed along with G(i).

```
procedure CONCENTRATE(G,r)
{ concentrate the G-records on the r-values }
{ r is a field of G }
for b:= 0 to logN-1 do
        if (r(i) is defined) and (r(i)_b<>i_b) then
            G(i^(b)) <- G(i);
end; { concentrate }
```

3. The DISTRIBUTE(G,d) Operation

The distribute operation performs the routing of the R-records in reverse the direction of the routing of the ranking operation.

```
procedure Distribute(G,d)
{ concentrate the G-records on the d-values}
{ d is a field of G satisfies the property stated
     in the definition of DISTRIBUTE}

for b:=logN-1 down to 0 do
        if (d(i) is defined) and (d(i)_b<>i_b) then
             G(i^(b)) <- G(i);
end; { concentrate }
```

4. The GENERALIZE operation

GENERALIZE is similar to DISTRIBUTE except that it must decide when to make a copy of the generalized records at the PE's along the routing. Nassimi and Sahni [NS1] show that this can be done in constant time in each of the logN phases (similar to DISTRIBUTE) by some comparisons of the ids of the PE's and the d-values on which we generalize.

The complexities of the above algorithms is O(logN) time, O(N) processor where N is the number of processors involved in the computation. Nassimi and Sahni [NS1] also give similar algorithms with the same time and processor complexities for the above operations on the PSC.

The time complexities of the RAR and RAW operations on these two SIMD architectures are therefore $O(\log^2 N)$. In the next section we shall develop new SIMD architectures to take advantage of these results.

4.4. New Interconnection Networks to Support Polylog Algorithms

Using the above simulation of CR and CW by RAR and RAW on the CCC or the PSC in $O(\log^2 N)$ SIMD time we can run the component labeling algorithms in Section 2 in $O(\log^3 N)$ SIMD time using N SIMD PE's. Although the local neighborhood computations can be done using the RAR and RAW on the CCC or the PSC, we would like to have in the SIMD computer the links of a MCC containing the image. The two proposed SIMD interconnections are very simple: the first one is a CCC with an embedded MCC; the second is a MCC with additional 3 PSC links (the exchange, shuffle and unshuffle links of the PSC) per PE. We call the former a **ECCC** (Embedding Cube-Connected Computer) and the latter a **MPSC** (Mesh and Perfect Shuffle Computer). Each PE possesses two alternative ids: its coordinates and the corresponding shuffled row-major index (see Section 2) or block index (see below). The conversions between these types of ids are just simple bit operations which can be hardwired into the system.

In both cases, for practical image sizes, compared to the MCC, the numbers of additional links are reasonably small and compared to the PC and the MOT there is no increase in the number of processors. Links are cheaper to build than processors. The links of the CCC in the ECCC and the added links of the PSC in the MPSC can be used for the simulations of the PRAM CR and CW operations by the RAR and RAW operations of SIMD computers. Local operations can be done in O(1)

time using the mesh part of the interconnection network. Hence the PRAM algorithms in section 2 can be systematically translated into the ECCC and MPSC algorithms and will have (SIMD) $O(\log^3 N)$ time and $O(N)$ processor complexities.

4.4.1. Embedding of the MCC in the CCC Using Block Indexing

If we can index the PE's of a MCC in such a way that every two 4 neighbors' ids differ in only one bit position then we can embed it in the CCC in a natural way. We can build the CCC architecture so that each node knows which of the logN links in the CCC are links to its 4 neighbors in the MCC. Alternatively, we devise an indexing scheme called block indexing which satisfies the property stated above.

In the block indexing scheme, we assign indices to the elements of an array as follows.

- We assign the 0th bits of the ids along each row of the array according to the sequence

0 1 1 0 0 1 1 0 0 1 1 0 0 ....

- We assign the 1st bits of the ids along each column of the array according to the sequence

0 1 1 0 0 1 1 0 0 1 1 0 0 ....

- We assign the 2nd bits of the ids along each row of the

array according to the sequence

0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 ....

- We assign the 3rd bits of the ids along each column of the array according to the sequence

0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 ....

and so on (see Figure 22).

This process produces an indexing such that the array is hierarchically subdivided into 2^b-blocks (as defined before) where 0<=b<N and neighboring array elements have indices differing in exactly one bit position. Furthermore, the conversions between the (i,j) coordinates and the block indices can be done in parallel efficiently by boolean circuits with constant depths and O(logn) widths.

For example, consider the computation of the block index of the element (i,j) of the array. The (2s)th bit I_{2s} of the index is computed from the binary representation of i as follows.

I_{2s} = the exclusive or of the sth

and (s+1)st (if exists) bit of i.

The (2s+1)st bit I_{2s+1} of the index is computed from the binary representation of j as follows.

I_{2s} = the exclusive or of the sth

and (s+1)st (if it exists) bit of j (see Figure 21).

We can run Algorithm 3 to set up the links of the embedded MCC in the EMCC.

ALGORITHM 3. Embedding the block indexed MCC in the CCC.

{let **B_to_C**(I) be the function that converts a block index to the corresponding coordinates index and **C_to_B** its inverse}
{PE(i) executes the following program:}

(i,j) <- **B_to_C**(I);
{(i,j) is the coordinates of the corresponding pixel}
if j-1>=0 **then**
the link to **C_to_B**(i,j-1) is the West link
else there is no West link;
if j+1<n **then**
the link to **C_to_B**(i,j+1) is the East link
else there is no East link;
if i-1>=0 **the**
the link to **C_to_B**(i-1,j) is the North link
else there is no North link;
if i-1>=0 **then**
the link to **C_to_B**(i-1,j) is the South link
else there is no South link;

4.4.2. Adding the MCC Interconnection Network to the PSC

The MCC can be indexed by the shuffle row-major or the block indexing scheme.

4.4.3. Simulation of Other SIMD Architecture by the ECCC and the MPSC

We use the same idea as in the arrangement of m processors to function as 2m-1 nodes of a complete binary tree. Each simulation of a level to level data movement in the MOT or in the PC costs $O(\log^2 N)$ time for RAR or RAW.

5. CONCLUSION

We have shown that a class of PRAM algorithms, including those in Section 3, can be run on the SIMD computers ECCC and MPSC with $O(\log^2 N)$ penalty for the simulation of CR and CW and no increase in the number of processing elements. These proposed SIMD interconnection schemes support the operations of component labeling, contour filling and component shrinking efficiently in $O(\log^3 N)$ time. The existing SIMD component labeling algorithms (for the MCC, the MOT and the PC) will probably get 4 times more complicated when 8-connected components are to be labeled (they only give solutions for the 4-connected component labelling) and do not give as a result an algorithm for contour filling.

The algorithms given in this paper may be generalized to the case of 3D binary images. Other operations such as finding the convex hull, detecting digital straightness can be solved efficiently in SIMD polylog time once polylog algorithms for them on the PRAM models are found.

We also arrive at a paradigm for the modular construction of SIMD architectures for low level image processing operations. The basic interconnection network is that of a 2 dimensional MCC where there is a PE for each pixel. We can view the PC (MOT, resp.) as a combination of MCC's and additional processing elements and connections in the form of a quaternary tree (binary trees, resp.).

In this light, we have demontrated the efficiency and simplicity of the architectures obtained by adding links to the MCC to form the ECCC and the MPSC. Efficient local operations can be done on the MCC interconnection part and global communications (RAR and RAW) on the CCC or PSC parts.

In image processing applications, the bottleneck the simulation of CR and CW by RAR and RAW is the sorting operation because other local operations require only constant time (in the two proposed architectures). It is interesting to note that the sorting operation which seems unnatural in biological vision systems plays an important role in our solution strategy. In a recent paper, Reif and Valiant [RV] give an O(logN) time randomized parallel sorting algorithm on a linear-sized SIMD architecture similar to the CCC. Using this randomized parallel algorithm and the routing method of Nassimi and Sahni [NS1], we can have randomized parallel algorithms for RAR and RAW data routing with O(logN) time complexity. Hence the SIMD algorithms developed in this paper have similar randomized versions with $O(\log^2 N)$ time complexity. Alternatively, (using the above paradigm for the modular construction of SIMD computers) we can add to the ECCC or the MPSC links of the with O(logN) time sorting networks [BP,LE] to obtain SIMD architectures with O(logN) time RAR and RAW.

Another problem of interest is to derive parallel algorithms with optimal speed-ups, i.e. parallel algorithms whose product of the

time and processor complexity terms is of order O(N), the time complexity of the optimal sequential algorithm for component labeling obtained by applying the depth first search algorithm to the induced graph of the image.

6. SUMMARY OF CONTRIBUTIONS

The followings are the contributions of this thesis.

1. The Label Propagating Algorithm (O(logN) time, O(N) processor on the PRAM model), a new parallel version of the sequential component labeling algorithm by the raster scanning method.

2. A paradigm for the simulation of a large class of PRAM algorithms which includes the Tree Merging and Label Propagating Algorithms on the SIMD model with $O(\log^2 N)$ time penalty where N is the input size and the number processors involved in the computation.

3. The block indexing used to embed the MCC in the CCC.

4. A paradigm for the modular construction of interconnection networks of SIMD computers (which are efficient for image processing applications).

5. A number of efficient SIMD architectures that support efficient component labeling, contour filling and component shrinking algorithms with O(N) processor complexity and $O(\log^2 N)$ or $O(\log^3 N)$ time complexities.

REFERENCES

[AS] Ahuja, N., Swamy, S., *Multiprocessor Pyramid Architectures for Bottom-Up Image Analysis, Multiresolution Image Processing and Analysis*, A. Rosenfeld (ed.), pp. 38-59.

[Ba] Batcher, K.E., *Sorting Networks and their Applications*, Proc. of the 1968 Spring Joint Comp. Conf. (Reston, Va., April), AFIPS, 307-314.

[BF] Bilardi, G., Preparata, F.P., A Minimum Area VLSI NetWork for O(logn) Time Sorting, IEEE Trans. on Computers, TC 34, No.4, April 1985.

[Bu] Burt, P.J., *The Pyramid as a Structure for Efficient Computation*, Multiresolution Image Processing and Analysis.

[BV] Bar-On, I., Vishkin, U., *Optimal Parallel Generation of a Computation Tree Form*, ACM Trans. on Prog. Lang. and Systems, vol.7(2), April 1985.

[Du] Dubitzki, T., Wu, A., Rosenfeld, A., *Parallel Computation of Contour Properties*, University of Maryland TR- 848, Dec. 1979.

[Fr] Freeman, H., *Computer Processing of Line-Drawing Images*, Computing Surveys, vol. 6, no.1, March 1974.

[Ga] Gazit, H., *An Optimized Randomized Parallel Algorithm for Finding Connected Components in a Graph*, 1986 IEEE FOCS.

[HR] Hanson, A.R., Riseman, E.M., *Preprocessing Cones: a Computational Structure for Scene Analysis*, TR-74C-7, U.of Mass., Amherst, MA.

[HCS] Hirschberg, D.S., Chandra, A.K., Sarwate, D.V., *Computing Connected Components on Parallel Computers*, CACM, August 1979, 22(8).

[KR] Kak, A., Rosenfeld, A., *Digital Image Processing*, Academic Press.

[KE] Kumar, P.V.K., Eshaghian, M.M., *Parallel Geometric Algorithms for Digitized Pictures on Mesh of Trees*, IEEE 1986 International Conference on Parallel Processing.

[Kn] Knuth, D., *The Art of Computer Programming*, vol.3, *Sorting and Searching*.

[Le] Leighton, T., Tight Bounds on the Complexity of Parallel Sorting, IEEE Transaction on Computers, TC 34, No.4, April 1985.

[MS1] Miller, R., Stout, Q.F., *The Pyramid Computer for Image Processing*, 1984 IEEE International Conference on Computer Vision and Pattern Recognition.

[MS2] Miller, R., Stout, Q.F., *Geometric Algorithms for Digitized Pictures on a Mesh-Connected Computer*, IEEE Trans PAMI, PAMI-7(1985).

[MS3] Miller, R., Stout, Q.F., *Data Movement Techniques for the Pyramid Computer*, SIAM Journal of Computing, 16(1), February 1987.

[NS1] Nassimi, D., Sahni, Sartaj S., *Data Broadcasting in SIMD Computers*, IEEE Transactions on Computers, TC-30(2), February 1981.

[NS2] Nassimi, D., Sahni, S., *Finding Connected Components and Connected Ones on a Mesh Connected Parallel Computer*, SIAM Journal of Computing, 9(4), Nov. 1980.

[RV] Reik, J., Valiant, L., *Logarithmic Time Sort for Linear Size Networks*, JACM, Feb. 1987, 34(1).

[Ro1] Rosenfeld, A., *Connectivity in Digital Pictures*, JACM 17(1), 156-160 (1970).

[Ro2] Rosenfeld, A., *Some Useful Properties of Pyramids*, Multiresolution Image Processing and Analysis, pp. 2-5.

[Sa] Samet, H., *The Quadtree and Related Hierarchical Data Structures*, Computing Surveys, 16(2), June 1984.

[St] Stone. H., *Parallel Processing with the Perfect Shuffle*, IEEE Transactions on Computers, C-20(2), Feb. 1971.

[SV] Shiloach Y., Vishkin U., *An O(logn) Parallel Connectivity Algorithm*, Journal of Algorithm, vol. 3, pp. 57-67, 1982.

[Si] Siegel, H.J., *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*.

[St] Stout, Q.F., *An Algorithmic Comparison of Meshes and Pyramids*, Evaluation of Multicomputers for Image Processing, Academic Press, 1986.

[Ta1] Tanimoto, S.L., *Hierarchical Approaches to Picture Processing*, Ph.D. dissertation, Dept. EE. and CS, Princeton 1975.

[Ta2] Tanimoto, S.L., *Sorting, Histogramming and Other Statistical Operations on a Pyramid Machine*, Multiresolution Image Processing and Analysis, pp.136-147.

[TK] Thompson, C.D., Kung, H.T., *Sorting on a Mesh-Connected Parallel Computer*, CACM, 20(4), April 1977.

[T] Tucker, L., *Labeling Connected Components on a Massively Parallel Tree Machine*, IEEE Conference on Computer Vision and Pattern Recognition, June 1986.

[U] Uhr, L., *Layer "Recognition Cone" Networks That Preprocess, Classify and Describe*, IEEE Trans. on Computers, TC. 21, 758-768.

[WBR] Wu A., Bhaskar, S.K., Rosenfeld, A., *Parallel Processing of Region Boundaries*, University of Maryland TR-1573, Nov. 1985.

APPENDIX 1. THE TREE-MERGING ALGORITHM

We assume the followings:

1. The binary image is stored in the array B in the shared memory such that the cell B(i,j) contains the value of the pixel (i,j), 0<=i,j< n.

2. We associate the processor P(i,j) with each pixel (i,j), 0<=i,j< n.

3. Furthermore, we need two variable s and s' and an array Q(i,j) for termination detection , 0<= i,j < n. During the algorithm, Q always satisfies:

Q(i,j)= s if after the second step of the sth iteration, a new vertex points to (i,j) and Q(i,j)< s otherwise.

We use the coordinates (i,j) as possible labels of the connected components. The output of the algorithm is an array L containing the component label of the pixel (i,j), 0<= i,j< n. Algorithm 1 is a detailed description of the component labeling for the binary image B. Figure 12 shows how Algorithm 1 works on a binary image.

ALGORITHM 1.

BINARY COMPONENT LABELING BY THE TREE-MERGING METHOD

0. **Initialization:**

s <- 1; s'<- 1;

for each "vertex" (i,j) **do in parallel**

begin

allocate processor P(i,j) to "vertex"(pixel) (i,j);

L(i,j) <- (i,j); Q(i,j) <- 0;

end;

while s = s' **do** {step s}

for each "vertex" (i,j) **do in parallel**

begin

1. **CR** L(L(i,j)) **into** L(i,j); {skipping on the array L}

if L(i,j) was changed **then CW** s **to** Q(L(i,j));

2. **if** B(i,j)= 1 **and** L(i,j) was not changed in step 1 **then**

{(i,j) points to a root}

begin

let (u,v) be such that

L(u,v) = **min** { L(x,y): L(x,y)<L(i,j) and (x,y) is a neighbor of (i,j)};

if (u,v) exists **then**

{the tree (u,v) points to a vertex smaller than the root of the

tree of (i,j), hook the tree of (i,j) onto the tree of (u,v)}

```
            begin
                CW L(u,v) to L(L(i,j));
                {many such (i,j) may write concurrently
                into L(L(i,j)), however only one will succeed}
                CW s to Q(L(u,v));
            end;
        end;

    3. if L(i,j)=L(L(i,j)) and Q(L(i,j))<s then
        {(i,j) points to a stagnant root}
        begin
          {the tree of (i,j) is stagnant}
          let (u,v) be such that
          L(u,v) = min { L(x,y): L(x,y)<>L(i,j) and (x,y) is a neighbor of (i,j)};
          if (u,v) exists then
          {try to hook the tree of (i,j) onto the tree of (u,v)}
          CW L(u,v) to L(L(i,j));
        end

    4. CR L(L(i,j)) into L(i,j); {do a second skip}

    5. if Q(i,j)=s then CW s+1 to s;
       s' <- s'+1;
       { if there is a non-stagnant tree, s' is updated otherwise s' will be less than s
         the algorithm terminates}
end {for};
```

Note that in step 2 and 3, each processor (i,j) does the works of the processors associated with its outgoing edges in Shiloach and Vishkin's algorithm. Instead of the arbitrary CW by the processors associated with the edges outgoing from the vertex (i,j), the minimum of the write values is computed and written by the processor (i,j).

In steps 1 and 2 of Algorithm 1, if there is a change at a node (it points to a new node or a new node points to it) then the iteration number is written (concurrently) into its Q variable. In step 5, if there has been no change in steps 1, 2, 3 and 4 then s is not updated and the testing at the while head in the next iteration results in (s<s') being true and the while loop terminates.

APPENDIX 2. COMPUTING THE BOUNDARY CHAINS IN PARALLEL FROM 3X3 NEIGHBORHOODS

If there is no singular pixel then it is easy to orient the edge of G to turn it into the boundary chains of the image. In this case, every boundary pixel has degree 0 or 2 in the induced graph C of the boundary of the image. In the latter case, only local information on the 3x3 neighborhood of each boundary pixel is needed to assign consistently the orientation of the edges of C to form the boundary chains (which are disjoint).

In the general case, we obtain the boundary chains from G by assigning to each edge of G 0, 1 or 2 directed edges. The graph induced by the boundary chains is Eulerian. A pixel can appear in at most 4 distinct boundary chains. Difficulties arise because of (using the analogy in Section 3.2) thin lines or "choke points" in the image. We assign to the boundary pixels the links in the boundary chain as follows. We "magnify" the image so that pixels now have substantial size and thin lines or choke points have sufficient mass so that we can assign the orientation in the simple case. A pixel appearing in a thin line or being a choke point is split up into a sufficient number of nodes for the distinct boundary chains it lies on. This is sufficient to ensure that the outside boundary chain of every component is computed as a closed path and each inside boundary chain surrounds only one hole. Figure 23 (24, resp.) shows how this is done for a number of cases of 3x3 4-(8-, resp.) neighborhoods (the

assignment of incoming and outgoing links is for the center 1-pixel). We match the incoming and outgoing links as indicated.

APPENDIX 3. THE LABEL PROPAGATION ALGORITHM.

ALGORITHM 2. BINARY COMPONENT LABELING BY LABEL PROPAGATION

PE(u,v): {j: a controller's variable; B(u,v), 0<=u,v<n: the binary image;

BND(u,v), W(u,v), N(u,v), L(u,v): variables associated with pixel (u,v);

The algorithm terminates with L(u,v) containing the component label of pixel (u,v).

}

0. L(u,v) <- (u,v);

1. Construct boundary chains for regions of 1-pixels;

{Now each PE corresponding to a pixel on a boundary

chain has BND(u,v) set to true (and false otherwise) and

N(u,v) set to the id of the next PE on the same chains}

{propagate min labels along boundary chains}

for j:= 1 **to** 2k **do if** BND(u,v) **then**

begin

2. **CW** L(u,v) **to** L'(N(u,v));

{pass the label (u,v) to the next PE, only EW is necessary }

3. L(u,v) <- **min**(L(u,v), L'(u,v));

{get the minimum label}

4. **CR** N(N(u,v)) **into** N(u,v);

{skip over the next PE}

end;

5. **Compute the inside boundary, outside boundary information for each boundary pixels.**

{Set up the stars representing links across holes}

6. **if** B(u,v)=1 and the East neighbor is a 0 and (u,v) is on an inside boundary **then** set W(u,v) to '(';
else
if B(u,v)=1 and the West neighbor is a 0 and
(u,v) is on an inside boundary **then** set W(u,v) to ')';
else set W(u,v) to '#';

Apply the generalized parentheses matching algorithm.
Now if W(u,v) contains a left parenthesis (left bank)
then N(u,v) contains the location of the matching
right parenthesis (right bank).

{set up the horizontal linked lists}

7. **if** B(u,v)=1 and East neighbor is a 1-pixel
then set N(u,v) to the id of the East neighbor
else N(u,v) <- (u,v);

8. **{find the end of the list}**
for j:= 1 **to** k **do**
if B(u,v)=1 **then**
CR N(N(u,v)) **to** N(u,v);

{skip over the next PE}

9. **if** B(u,v)=1 **then CR** L(N(u,v)) **to** L(u,v);

{read the label of the boundary pixel at the end of the list}

Figure 1. a. 4-neighborhood. b. 8 neighborhood.

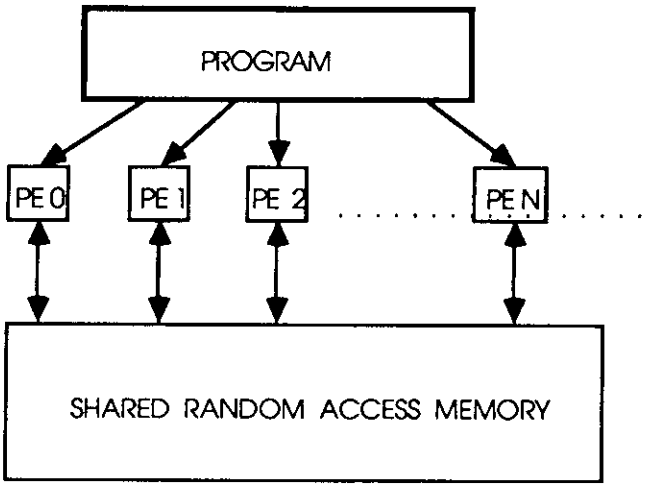


Figure 2. Diagram of a PRAM. .

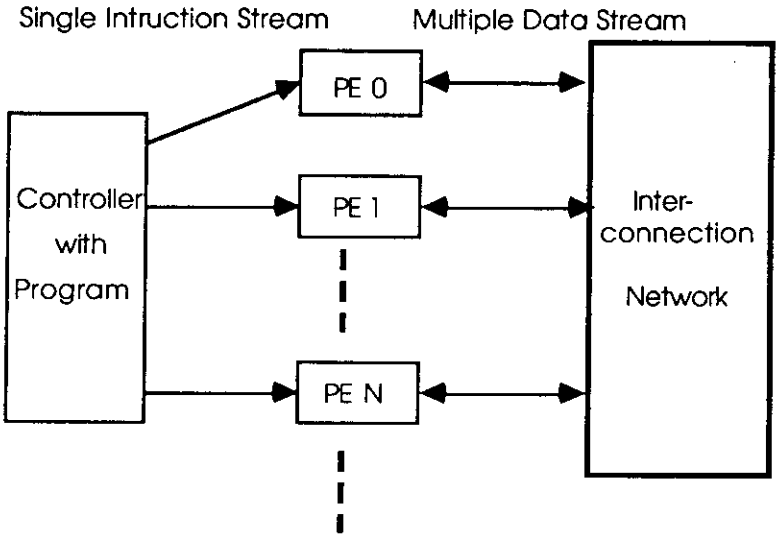


Figure 3. Diagram of a SIMD computer.

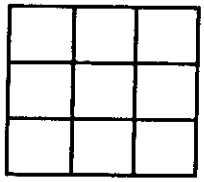


Figure 4. A mesh connected computer.

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Figure 5. Shuffled row-major indexing.

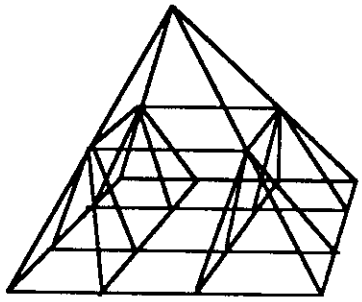


Figure 6 a. A pyramid computer with 4x4 base.

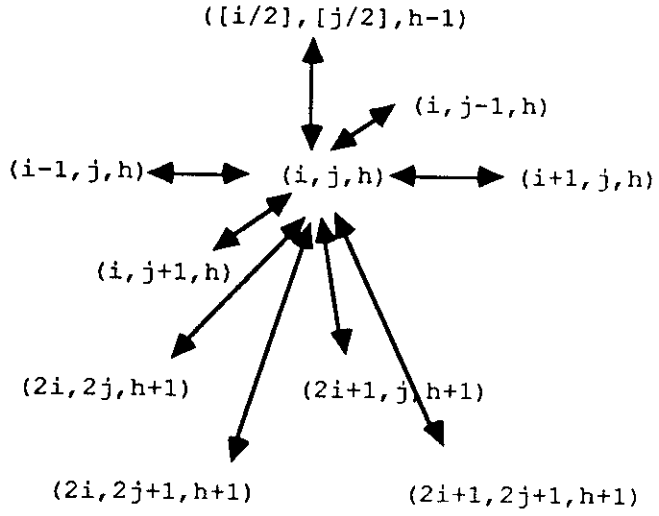


Figure 6b. Pyramidal Connections from PE (i, j, h).

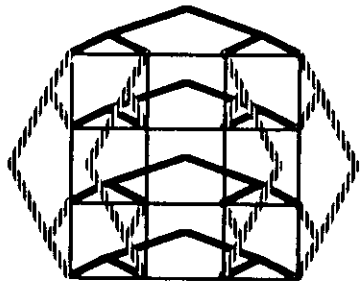


Figure 7. A 4x4 mesh ot trees computer.

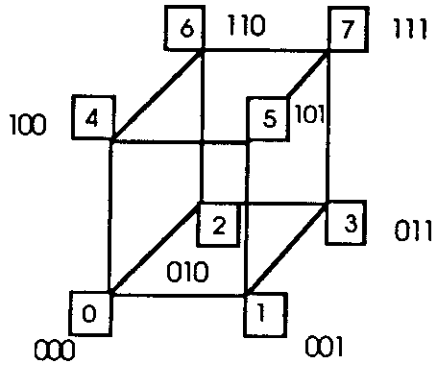


Figure 8. A cube-connected computer of size 8.

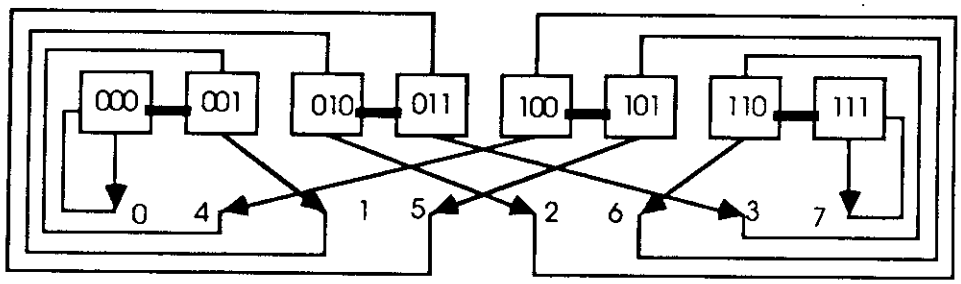


Figure 9. A perfect shuffle computer. The unshuffle links are opposite to the shuffle links.

shuffle link

exchange link

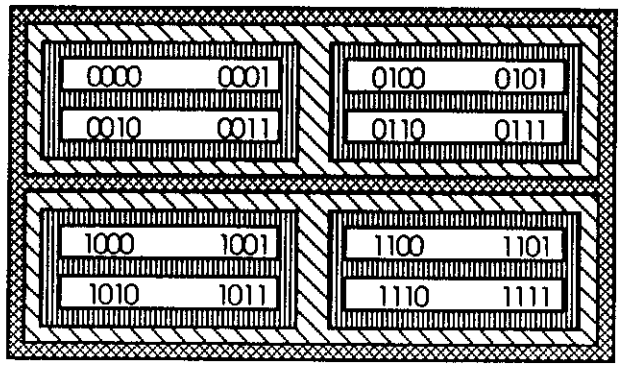


Figure 10. The nested 2^b -blocks of a shuffle row major indexed computer.

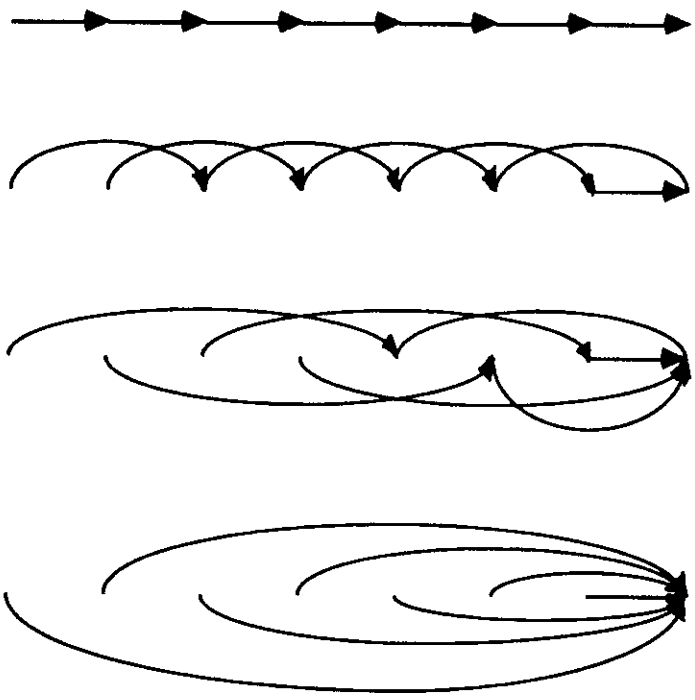


Figure 11. Pointer skippings on a linked list of length 8.

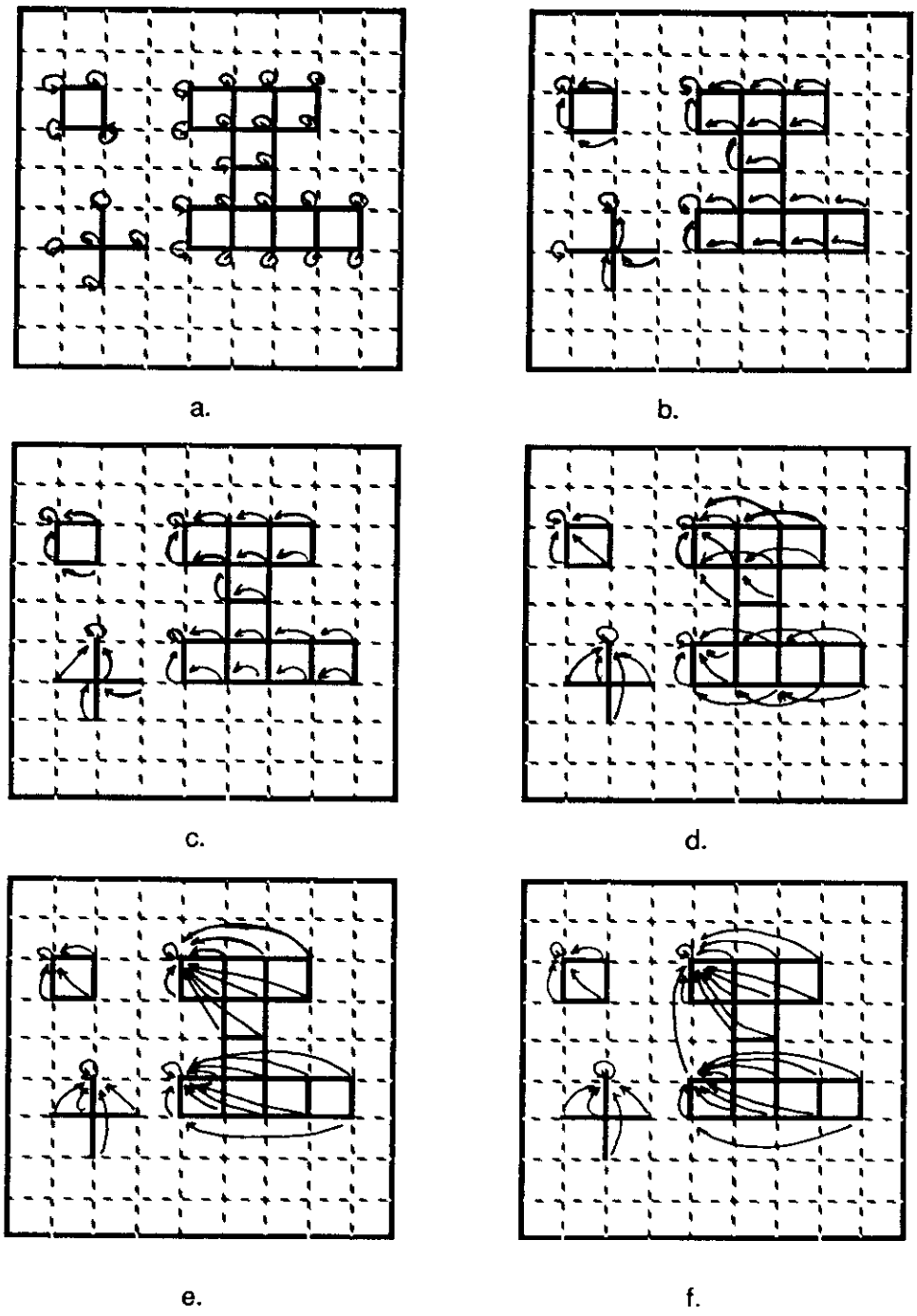


Figure 12. The Tree Merging Algorithm on an Image.
a. The image after step 1.1; b. Step 1.2, (1,6) is a stagnant root;
c. Step1.3, (1,6) hooks onto (2,5) d. Step 1.4;
e. Step 2.1; f. Step 2.2, (5,5) hooks (4,5) onto (4,2).

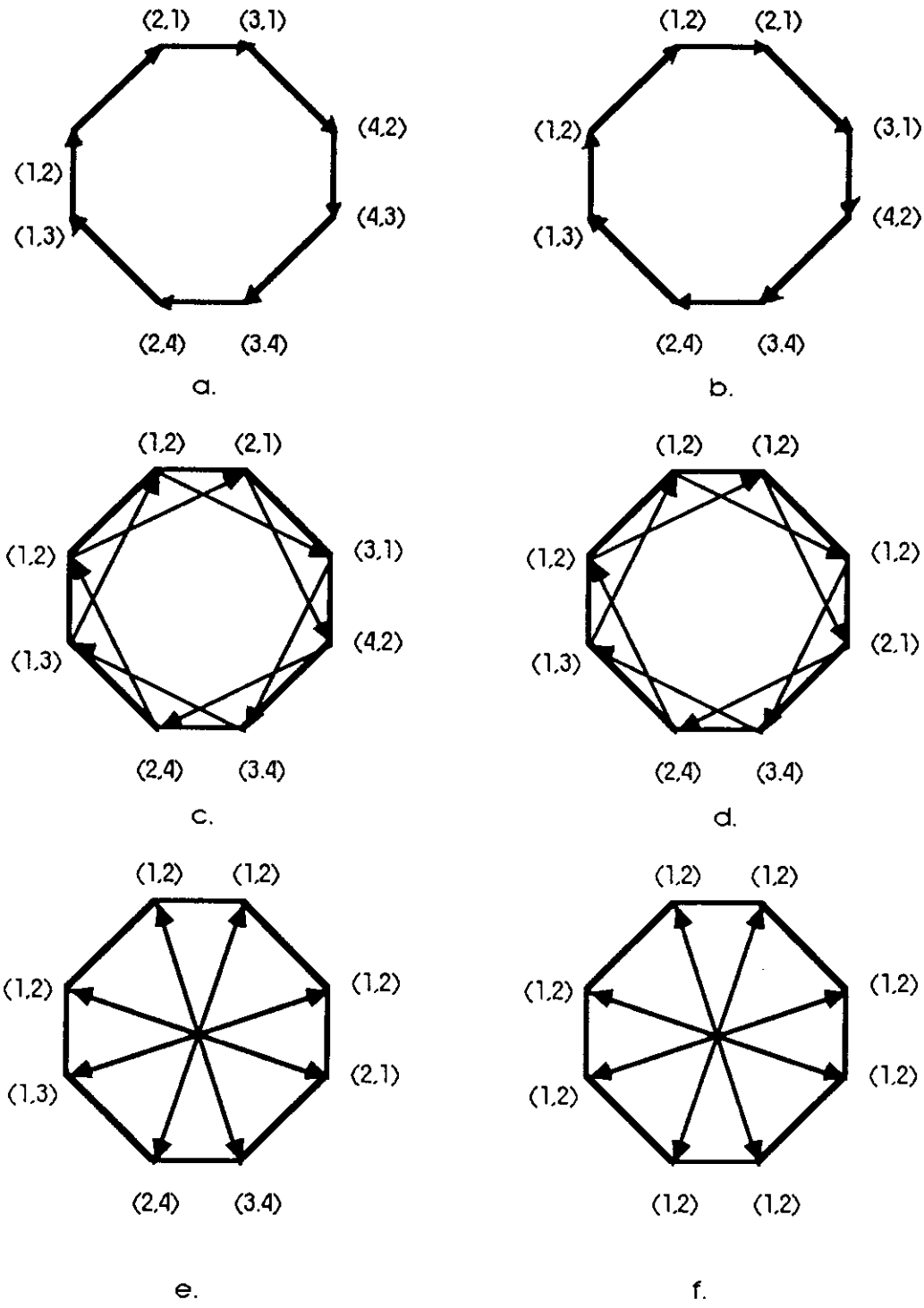


Figure 13. Propagating the minimum valuealong a cycle.
a. Original boundary chain. c, e. Pointer skipping.
b, d, f. Propagating the smaller values along the pointers

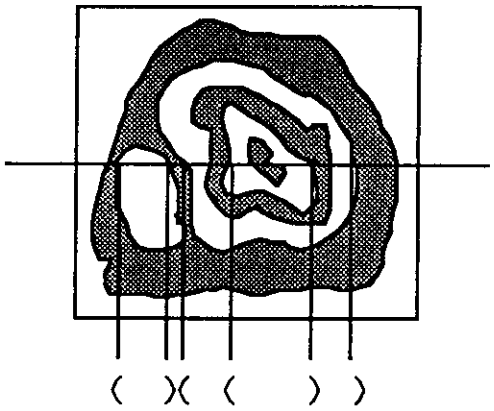


Figure 14 . Assignment of the balanced parenthesis expression.
The cuts corresponds to the strings
1...1*1...1*1...1, 1...1*1...1, 1...1 and the expression ( ) (( )).

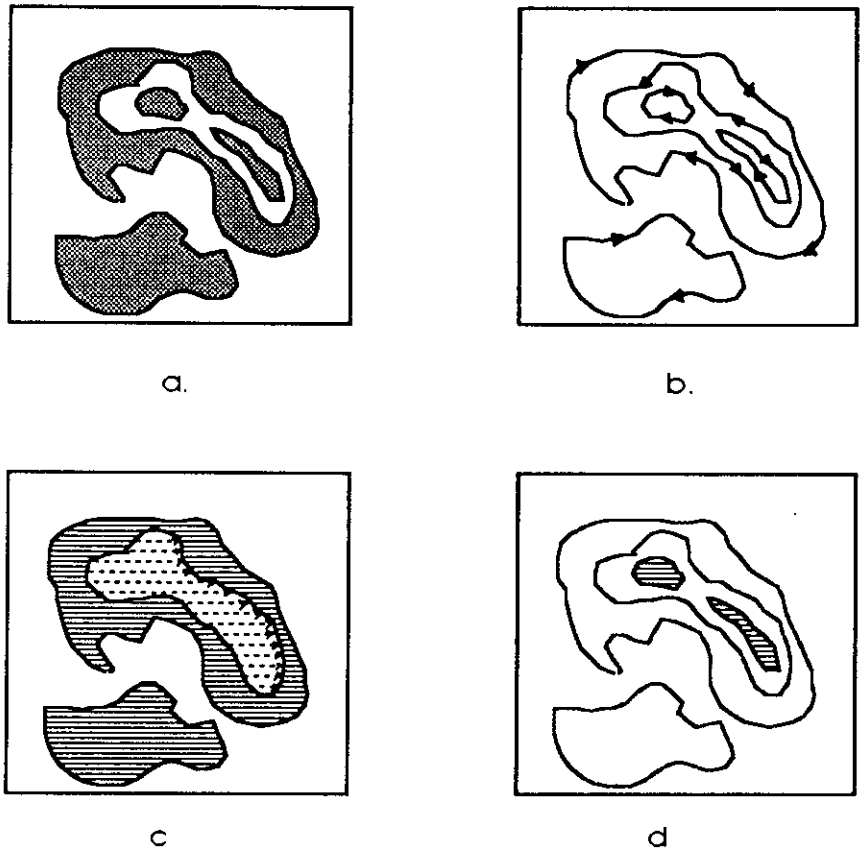


Figure 15. The Label Propagating Algorithms on an image.
a. Original Picture. b. Setting up the boundary chains.
c. and d. The horizontal linked lists of the connected components.

——— 1...1 section of the list ····· a star or a bridge over a hole

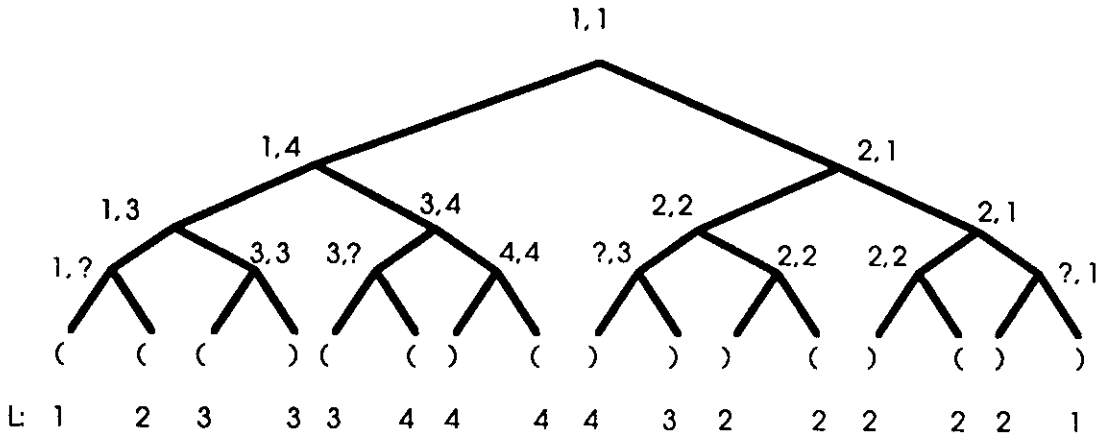


Figure 16. The binary tree structure for the string "((( ) (( ) ( ))) ( ) ( ))" and the L, MINLEFT, and MINRIGHT values ("?" denotes an undefined value).

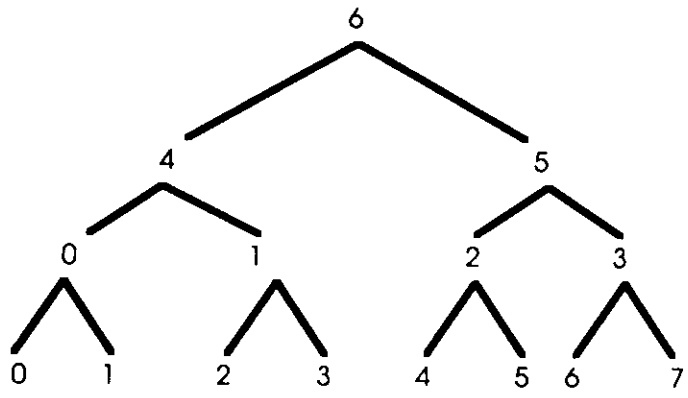


Figure 17. A binary tree with 2*8 - 1 nodes. The numbers shows the id numbers of the processors assigned to the nodes.

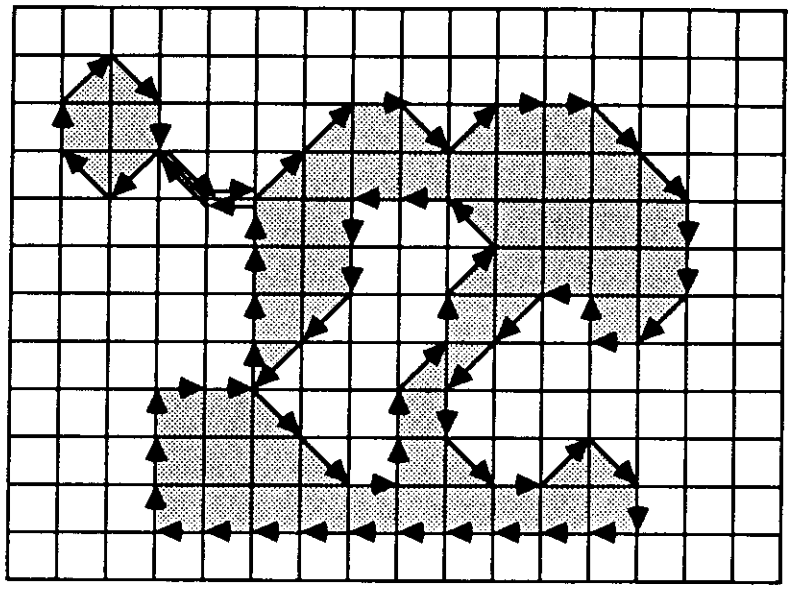


Figure 18. A binary image and its boundary chains.

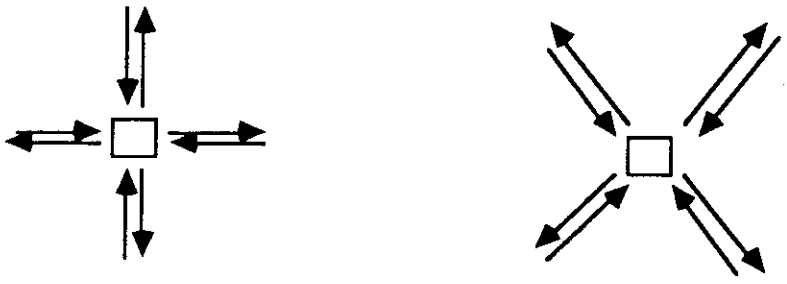


Figure 19. A pixel can appear many times in a boundary chain.

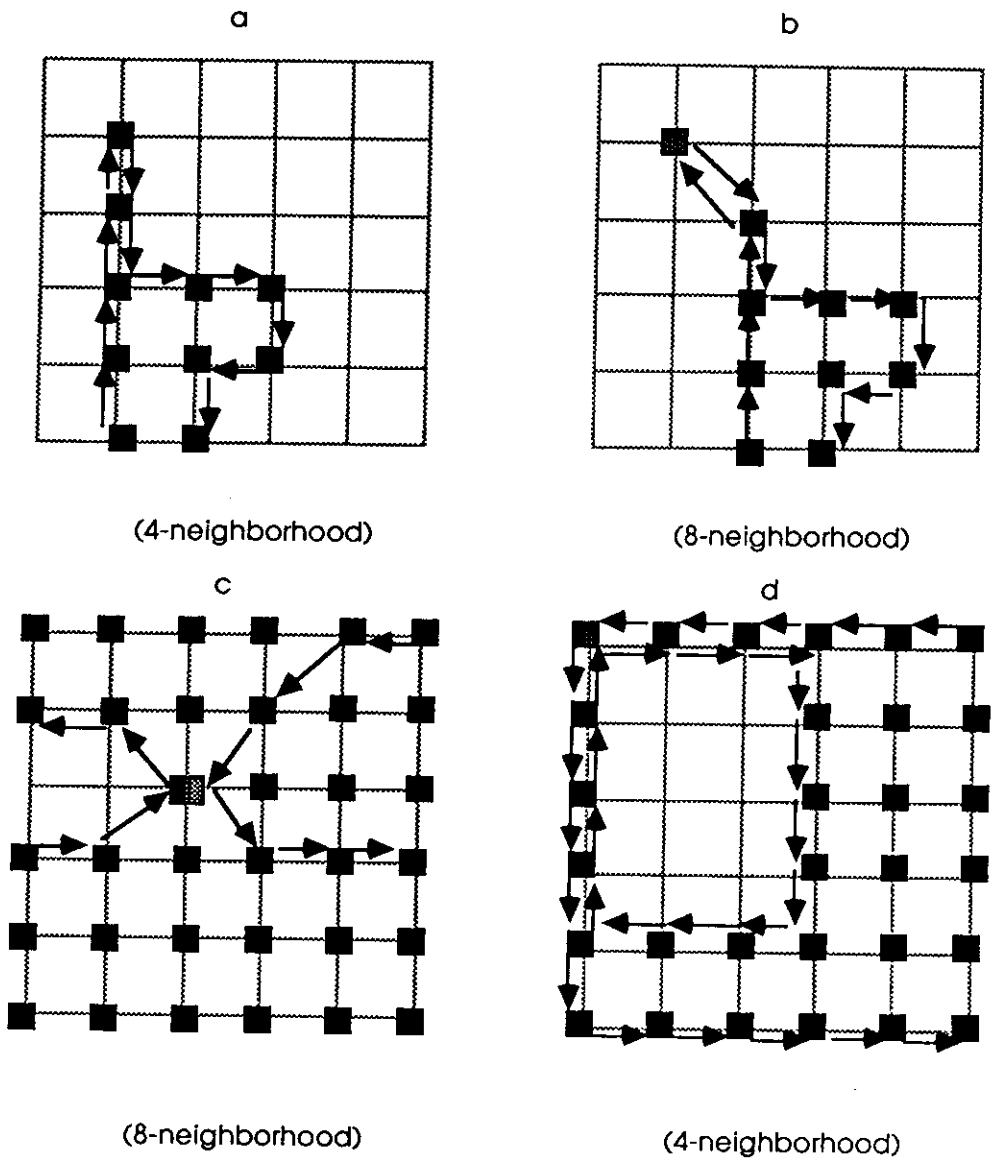


Figure 20. Some special cases to consider when the miminum node of the chain is on two boundary chains.

■ a minimum node on a boundary chain

In cases a and b, the minimum node must be on an outside boundary.
In cases c and d, the minimum node decides the orientations separately for each of the two touching boundary chains.

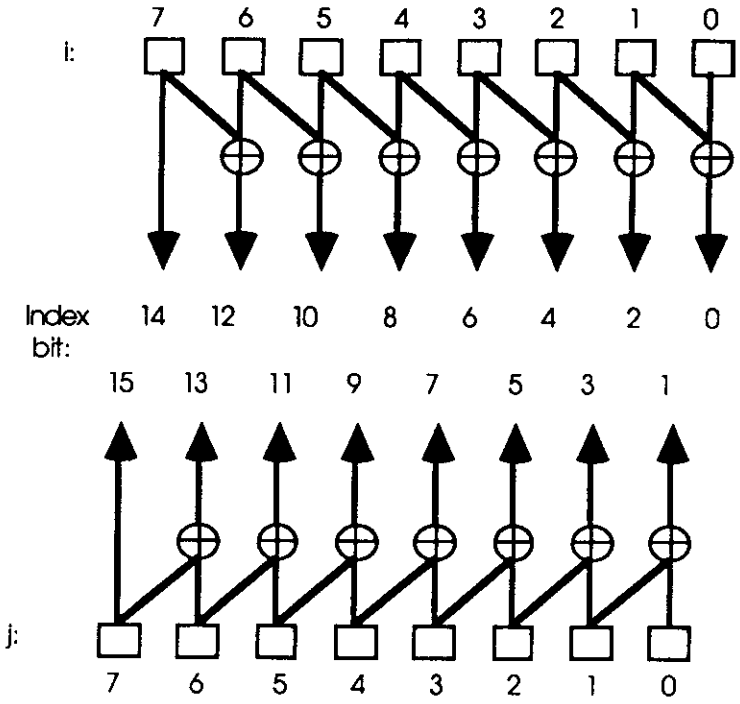


Figure 21. Conversion from the coordinates (i,j) to the corresponding block index for k=8.

⊕ Exclusive-Or Gate

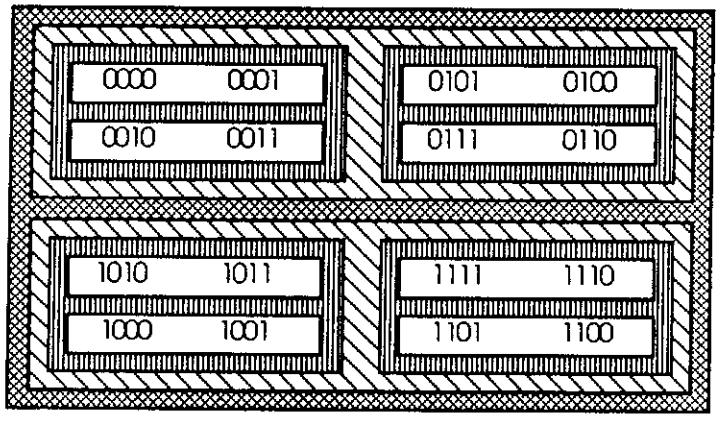


Figure 22. The nested 2^b -blocks of .a block indexed computer.

Figure 23. Some typical 4-neighborhoods and the assignments of boundary chain pointers. Other cases can be obtained by rotations.

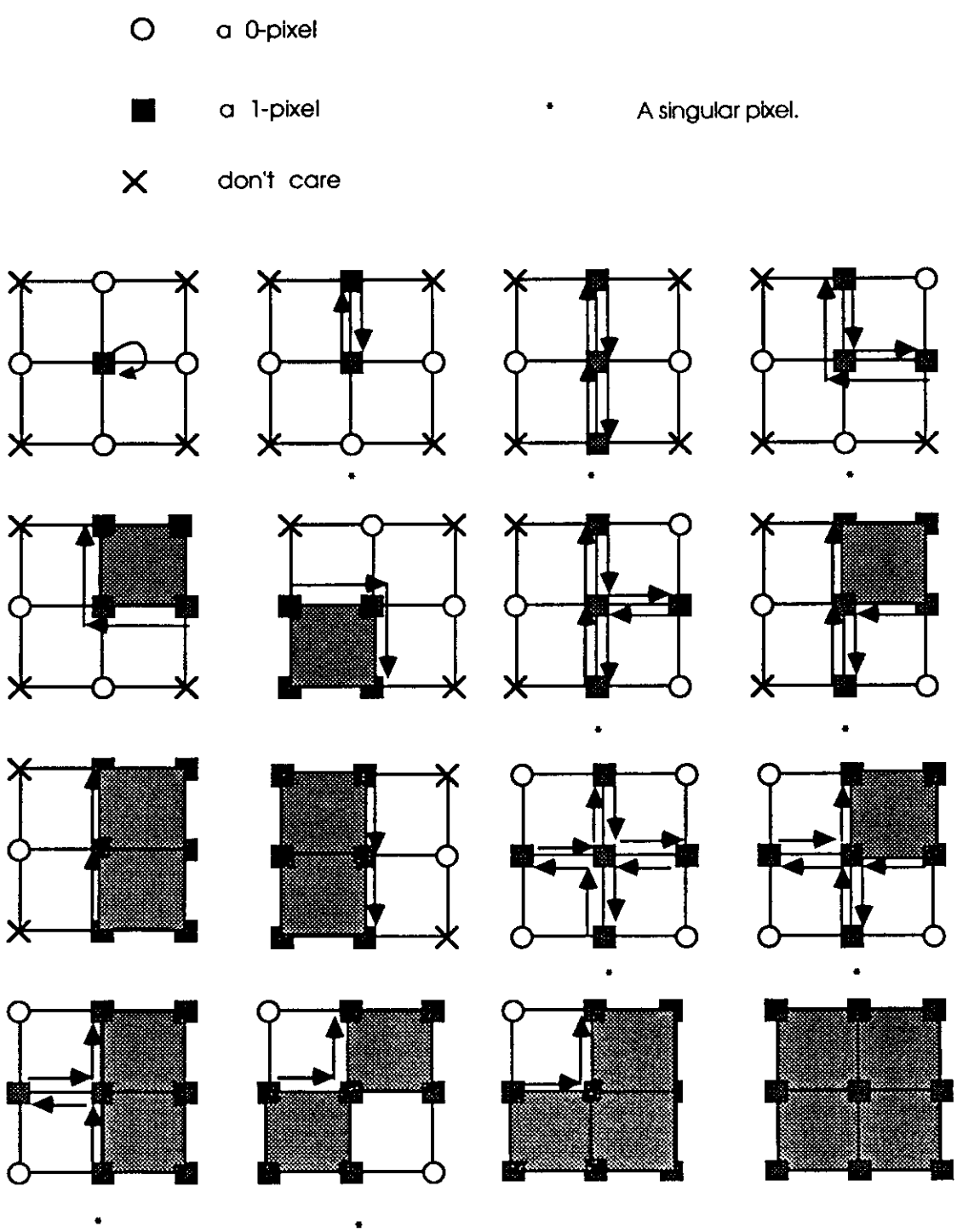


Figure 24. Some typical 8-neighborhoods and the assignments of boundary chain pointers. Other cases can be obtained similarly.

■ a 1-pixel • denotes a singular pixel

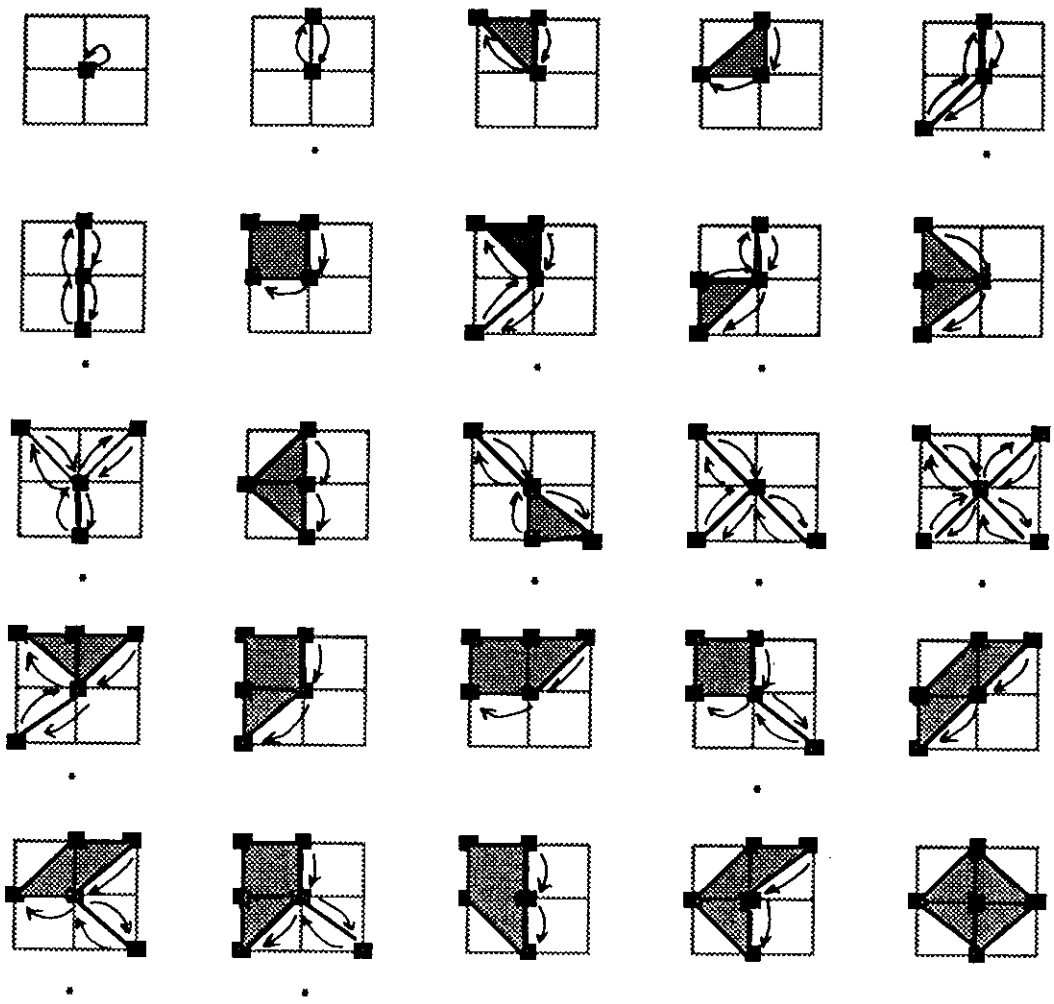


Figure 24 (continued)

■ a 1-pixel