

# Position-invariant hierarchical image analysis

James V. Mahoney

Xerox Palo Alto Research Center  
3333 Coyote Hill Road  
Palo Alto, CA 94304

David T. Clemens

Ascent Technology  
64 Sidney Street, Suite 380  
Cambridge, MA 02139-4136

## Abstract

*Spatial hierarchy is an appealing strategy for image analysis, but standard hierarchical methods give increasingly sparse, position dependent results with increasing level. This paper endows simple hierarchical techniques with position invariance by an architectural extension, broadening their scope with no increase in algorithmic complexity. The tree structure is extended into an exhaustive hierarchy containing a node at every image location at every level. This maps straightforwardly to existing parallel computers; one pass takes  $O(\log N)$  time on a hypercube and  $O(N)$  time on a mesh, for an  $N \times N$  image. We present novel algorithms for labeling connected components; detecting line features robustly; and computing distance transforms. The algorithms are exceedingly simple, usually optimal, and involve a minimum of storage and communication per node.*

## 1 Introduction

Because scene objects may be arbitrarily located and extended, processes for feature detection, grouping, segmentation, and figure/ground must integrate information over the entire image very rapidly. Given fine-grained parallelism, spatial hierarchy—which uniformly subdivides the image at a series of scales—is an appealing tool for making such *spatial integration* processes run fast, for the following reasons.

Hierarchy is parsimonious: wide ranging spatial integration processes are expressible using primitives just for arithmetic and parent-child communication. Purely hierarchical methods are uniform in space and scale, giving exquisitely concise algorithms for simple hardware. They produce large scale results in  $O(\log N)$  time, for an  $N \times N$  image, and, with pipelining, certain hierarchical processes can deliver large scale results on every cycle. Finally, the structure of a spatial hierarchy maps well to the communication patterns of spatial integration processes, so general, global data routing is unnecessary. Global routing and sorting algorithms on fine-grained parallel machines are quite complicated; avoiding them vastly simplifies an implementation. Further, hierarchical methods that eschew global ad-

resses apply even where numerical precision may be too limited to represent nodes or locations by unique identifiers.

Unfortunately, trees and pyramids yield increasingly sparse results—representing fewer regions—with increasing level.<sup>1</sup> This strong coupling between scale and localization is a significant impediment in several ways. *Position dependence*: straightforward hierarchical feature detection processes miss features that are “split” across region boundaries. *Algorithmic complexity*: to offset position dependence, more complicated algorithms are often used; e.g., merging feature fragments across region boundaries using the lateral connections in a pyramid. *Time complexity*: the use of such extra-hierarchical operations can also lead to much slower algorithms. *Limited scope*: scale-localization coupling restricts scope. For instance, image-to-image transformations that involve long-range pixel relations—such as the distance transform—do not map to trees or pyramids at all.

In this paper, we endow simple hierarchical techniques with position invariance by an architectural extension that broadens their scope with no increase in algorithmic complexity. We extend the tree into an *exhaustive hierarchy* containing a node for every discrete image location at every level. This maps straightforwardly and efficiently to existing parallel computers: one hierarchical pass takes  $O(\log N)$  time on a hypercube and  $O(N)$  time on a mesh, for an  $N \times N$  image.

We introduce exhaustive, hierarchical architecture in Section 2, and then make the case for it by presenting methods for several key spatial integration operations: defining local connected components (Section 3); labeling local and global components with their properties (Section 4); labeling line features robustly (Section 5); computing distance transforms (Section 6); and communicating between Voronoi neighbors (Section 6). These algorithms are exceedingly simple, usually optimal, and involve minimal storage and communication.

## 2 Exhaustive hierarchical architecture

**The binary image tree.** Figure 1 illustrates a binary hierarchical decomposition of an image into rectangular

<sup>1</sup>A pyramid is a tree structure augmented, at each level, with links between adjacent nodes.

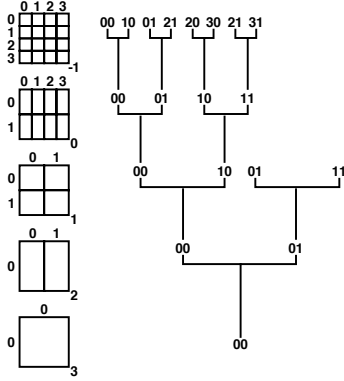


Figure 1: A binary image tree.

regions—termed a *binary image tree* [17], or *bintree* [20]. It is built on a square base array of width  $N$  and has  $2 \log N$  levels above the base. A region at level  $l$ , a *parent*, is the union of two *child* regions at level  $l - 1$ . A parallel hierarchical process loops upward (or downward) through the hierarchy, at each level applying the same arithmetic and logical operations at every node in parallel, and storing the results in the nodes. This process maps directly to *single instruction, multiple data* (SIMD) parallel computing architectures. Given a processor per node per level, one pass up or down the hierarchy takes  $O(\log N)$  time.

The methods in this paper apply equally to a *quadtree* architecture [18], but we prefer bintrees for they allow more concise algorithms.

**The binary image jungle.** A tree structure may be extended into a hierarchical network whose local connectivity is treelike, but which exhibits no “narrowing” as level increases. Each level in the tree is replaced by an  $N \times N$  array of nodes. A node still has exactly two children, but now each node is the first child of one parent node and the second child of another. This is illustrated in one dimension in Figure 2. The term *binary image jungle* (BIJ) extends the “tree” metaphor to capture the dense, shared structure.

Let  $l$  range from 0 at the lowest parent level to  $H = 2 \log N$  at the highest. We map BIJ nodes to the image array such that a node at level  $l$  has first child offsets  $(0, 0)$ , and second child offsets  $(0, 2^{l/2})$  at even levels and  $(2^{\lceil l/2 \rceil}, 0)$  at odd levels. (The second child offsets at level  $l$  are denoted by  $\alpha_x(l)$ ,  $\alpha_y(l)$ .) One hierarchical pass still takes  $O(\log N)$ . Upward processes are unchanged by this extension; downward processes must now handle collisions.

Through reflexive Gray code mapping, an  $N \times N$  array can be mapped to a  $N^2$  node hypercube such that any two locations a power of two apart in both  $x$  and  $y$  are separated by just two communication wires [12] [20]. Therefore, if all BIJ nodes corresponding to a particular location are mapped to a single processing element in a hypercube computer, one

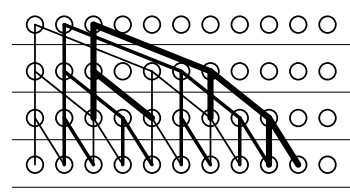


Figure 2: A binary jungle in one dimension. By sharing nodes and edges, a complete subtree is associated with every location at every level. Only three trees are shown.

exhaustive hierarchical pass takes  $O(\log N)$  time.

A BIJ can also be mapped to a mesh computer. To communicate between a node and its second child on a mesh, we sequentially shift the data through the  $2^l - 2$  intermediate processing elements, where  $l$  is the level of the parent. The communication time at level  $l$  of the hierarchy is  $O(2^l)$ . The communication time for a complete pass through the hierarchy is proportional to  $N + N/2 + N/4 + \dots + 1$ , which is  $O(N)$ . Therefore, one exhaustive hierarchical pass takes  $O(N)$  time on a mesh computer.

In an *in-place* algorithm on the BIJ, nodes at the same spatial location share the same storage, so the storage required per location is independent of the size of the image. All the computations in this paper allow in-place implementation, because parent results subsume child results at a given location. In-place methods conserve storage, but the computations cannot be pipelined across levels.

**Notation.** An exhaustive hierarchical algorithm is completely specified by three aspects: (i) the set of variables involved; (ii) how each variable is initialized at the base level ( $l = -1$ ); (iii) how each variable is updated at a parent level ( $l > -1$ ). The algorithm is perfectly uniform in both space and scale: exactly the same computations are done at every node, at every level  $l$ . In an upward (downward) process,  $l$  ranges consecutively from 0 ( $H - 1$ ) to  $H - 1$  (0).

$\mathbf{i}$  always denotes the input pixel value at the base.  $R$  denotes the region represented by a given node  $\eta$  at level  $l$ .  $\mathcal{P}(x)$  denotes the value of variable  $x$  in  $\eta$ .  $\mathcal{C}_1(x)$ ,  $\mathcal{C}_2(x)$  denote the values of  $x$  in the first, second child, respectively.  $\mathcal{C}_2(x) = 0$  if the second child does not exist, i.e., if its offsets fall outside the array bounds, in which case two alternate notations are used to specify boundary values:  $\mathcal{C}_2^w(x) = x$  and  $\mathcal{C}_2^b(x, b) = b$ . (Note that in an in-place implementation,  $\mathcal{C}_1(x)$  is a *no-op*, and  $\mathcal{C}_2(x)$  represents uniform translation.)

Our general scheme for specifying how a variable  $x$  is updated is illustrated by the following expression:

$$\mathcal{P}(x) \leftarrow f(\mathcal{C}_1(x), \mathcal{C}_2(x), \mathcal{C}_2(y), \mathcal{P}(z))$$

In this expression, the parent value of  $x$  is some function  $f$  of both child values of  $x$ , the second child value of  $y$ , and the parent value of  $z$ . In practise we use the following terse

form:

$$x : \mathcal{P} \leftarrow f(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_2(y), z)$$

That is,  $\mathcal{P}$ ,  $\mathcal{C}_1$ ,  $\mathcal{C}_2$  without arguments denote the value of the prefixed variable (i.e.,  $x$ ) in the parent, first child, and second child, respectively; other free variables denote values in the parent.

$\mathcal{T}(x, i, j)$  denotes the value of  $x$  at offsets  $i, j$ . When these offsets are out of the array bounds,  $\mathcal{T}(a, i, j) \triangleq 0$ .

The integers 1, 0 represent the booleans TRUE, FALSE. The following shorthand completes our notation:

$$\lambda(a, b) \triangleq \begin{cases} a & \text{if } l \text{ even} \\ b & \text{otherwise} \end{cases} \quad (1)$$

$$\gamma(a, b, c) \triangleq \begin{cases} b & \text{if } a \neq 0 \\ c & \text{otherwise} \end{cases} \quad (2)$$

$$[a = b] \triangleq \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

### 3 Defining local components

The methods of this section subdivide an image into regions each containing a single connected component. These regions serve as building blocks for global component labeling and as primitive units for measuring local properties. In the first method, the regions are squares and compact rectangles. In the second, they are segments of individual pixel rows or columns; the ‘‘runs’’ and run fragments that result were shown by Shafir [19] to support even faster component labeling than compact regions.

**2D single-component regions.** A region  $R$  is classified as containing (i) no black pixels (*vacant*); (ii) definitely one connected component (*valid*); or (iii) possibly more than one connected component, by hierarchically applying three rules: (i) the union of two adjacent vacant regions is vacant; (ii) the union of a vacant region and an adjacent valid region, or vice versa, is valid; (iii) the union of adjacent valid regions  $a, b$  is valid if some black pixel in  $a$  is adjacent to a black pixel in  $b$ . At the base, a white pixel is labeled vacant and a black pixel valid. Not every region containing a single connected component is labeled valid by this scheme, because it is local and hierarchical, whereas connectivity is global. (In particular,  $R$  will often not be labeled valid if the component it contains is not convex.)

The pixel adjacency constraint of rule (iii) can itself be established hierarchically. A black pixel is *right-connected* (*down-connected*) if it has a black four-neighbor to the right (below). Region  $R$  is right-connected (down-connected) if it has a right-connected pixel in its right (bottom) border. I.e., at even levels: (i)  $R$  is right-connected if either child is; (ii)  $R$  is down-connected if its second (bottom) child is. At odd levels: (i)  $R$  is right-connected if its second (right) child is; (ii)  $R$  is down-connected if either child is.

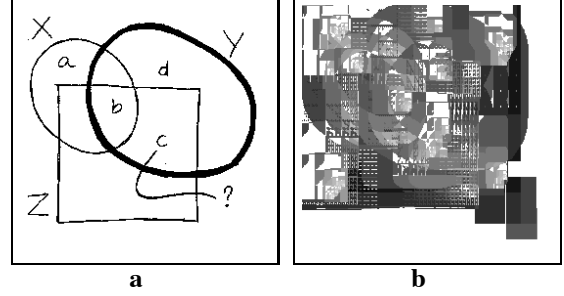


Figure 3: (a) Input. (b) Grey level at a pixel represents the maximum level at which the corresponding region contained a single connected component.

Formally, let  $u, x_d, x_r, v$  denote one-bit values representing whether  $R$  is vacant, down-connected, right-connected, valid, respectively. For  $l = -1$ ,  $u = \neg \mathbf{i}$ ;  $x_d = \mathbf{i} \cap \mathcal{T}(\mathbf{i}, 0, -1)$ ;  $x_r = \mathbf{i} \cap \mathcal{T}(\mathbf{i}, -1, 0)$ ;  $v = \mathbf{i}$ . For  $l > -1$ ,

$$u : \mathcal{P} \leftarrow \mathcal{C}_1 \wedge \mathcal{C}_2 \quad (4)$$

$$x_d : \mathcal{P} \leftarrow \lambda(\mathcal{C}_2, \mathcal{C}_1 \vee \mathcal{C}_2) \quad (5)$$

$$x_r : \mathcal{P} \leftarrow \lambda(\mathcal{C}_1 \vee \mathcal{C}_2, \mathcal{C}_2) \quad (6)$$

$$v : \mathcal{P} \leftarrow [\mathcal{C}_1(u) \wedge \mathcal{C}_2] \vee [\mathcal{C}_1 \wedge \mathcal{C}_2(u)] \vee [\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_1(x)] \quad (7)$$

where  $x = \lambda(x_d, x_r)$ .

**1D single-component regions.** To compute 1D components, the method above is modified so that it *idles* at alternate levels. Idling just passes on the results computed at the previous level: at even (odd) levels, idling skips all horizontal (vertical) communication. (In in-place implementation, idling simply involves incrementing  $l$  by 2.)

Consider the vertical case. (The horizontal case is symmetric.) Let  $u^v, x_d^v, v^v$  denote one-bit values representing whether  $R$  is vacant, down-connected, valid, respectively. For  $l = 0$ ,  $u^v = \neg \mathbf{i}$ ;  $x_d^v = \mathbf{i} \cap \mathcal{T}(\mathbf{i}, 0, -1)$ ;  $v^v = \mathbf{i}$ . For  $l > 0$ ,

$$u^v : \mathcal{P} \leftarrow \lambda(\mathcal{C}_1 \wedge \mathcal{C}_2, \mathcal{C}_1) \quad (8)$$

$$x_d^v : \mathcal{P} \leftarrow \lambda(\mathcal{C}_2, \mathcal{C}_1) \quad (9)$$

$$v^v : \mathcal{P} \leftarrow \lambda(\mathbf{t}, \mathcal{C}_1) \quad (10)$$

where

$$\mathbf{t} = [\mathcal{C}_1(u^v) \wedge \mathcal{C}_2] \vee [\mathcal{C}_1 \wedge \mathcal{C}_2(u^v)] \vee [\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_1(x_d^v)]$$

### 4 Local and global component labeling

**Labeling local component properties.** Local component properties like area and perimeter are given by simple hierarchical *accumulation* constrained to valid regions. Let  $a$  denote the variable for accumulation, and  $\oplus$  the operator for combining values. For  $l > -1$ ,

$$a : \mathcal{P} \leftarrow \gamma(v, \mathcal{C}_1 \oplus \mathcal{C}_2, \mathcal{C}_1) \quad (11)$$

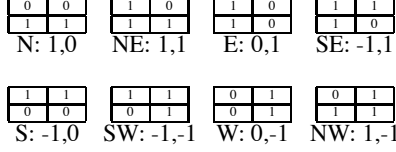


Figure 4: Edge pair patterns; the associated  $x, y$  displacements are used to compute local orientation.

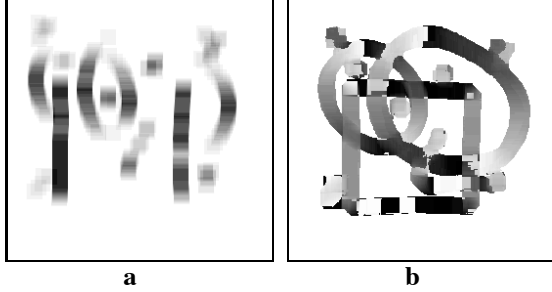


Figure 5: (a) The grey value at a pixel represents the W edge pair count in a corresponding  $w \times w$  region  $R$ . (b) The grey value at a pixel represents the local orientation measured in  $R$ .

For example, the area of each local component is given by initializing  $a$  to  $\mathbf{i}$ ,  $\oplus$  denoting addition. The perimeter is given by initializing  $a$  to  $e(\mathbf{i})$ , where  $e(\mathbf{i})$  is 1 at a black pixel adjacent to a white pixel. Similarly, we can measure the boundary orientation of a local component (Figure 5), since the  $x, y$  displacements of a straight boundary in  $R$  can be determined by counting the incidence of the  $2 \times 2$  pixel patterns shown in Table 4.

Accumulation over vertical 1D components is given by:

$$a_{\oplus}^v : \mathcal{P} \leftarrow \lambda(\gamma(v^v, \mathcal{C}_1 \oplus \mathcal{C}_2, \mathcal{C}_1), \mathcal{C}_1) \quad (12)$$

**Global label propagation.** Global propagation of the maximum or minimum over each connected component can be done by a sequence of accumulate-then-distribute operations. *Distribution* is a downward hierarchical process that copies values from valid parents to their children. Since a child has two parents, the values received from them must be combined somehow. Let  $\mathcal{P}'(x)$  denote the value of  $x$  in the *other* parent node of  $\eta$ , i.e., the node at level  $l + 1$  at offsets  $-o_x(l + 1), -o_y(l + 1)$ . Let  $a$  denote the variable for distribution.  $\oplus$  denotes the operation for resolving collisions. Let  $\mathbf{t} : \mathcal{C}_1 \leftarrow \mathcal{P} \vee \mathcal{P}' \vee \mathcal{P}'(v)$ . For  $H > l > 0$ ,

$$a : \mathcal{C}_1 \leftarrow \mathcal{P} \oplus \gamma(\mathbf{t}, \mathcal{P}', 0) \quad (13)$$

As an example, global connected components can be labeled with unique identifiers as follows. Initially, assign variable  $a$  a globally unique integer at each black pixel and zero at each white pixel.  $\oplus$  is max. Then we repeat the following steps until no two four-adjacent pixels in  $a$  have distinct values: (i) accumulate  $a$ ; (ii) distribute  $a$ ; (iii) swap

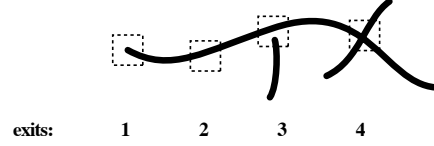


Figure 6: Line features are associated with a measure of local topology, the *exit count*.

values among four-adjacent pixels in  $a$ . The same process implements a *seed fill*, if  $\oplus$  is OR and  $a$  is initialized to 1 at the seed locations and 0 elsewhere.

The number of iterations,  $m$ , of this routine depends on the shape of the input figures, and on proximity relations among them, but is scale and position independent. ( $m$  has the same value that the most favorable positioning of a bin-tree over the image would give.)  $m$  may approach  $N^2$  in the worst case of a space-filling curve, but it typically ranges from one, for compact, somewhat isolated figures, to the tens, for elongated, complex figures. (E.g., the large component in Figure 3 (a) is labeled in eight steps.)

Unfortunately, the global *sum* over a connected component cannot be propagated by this process. A useful compromise is to label local connected components with the sum, and then propagate the maximal sum globally.

## 5 Line feature labeling

Curvilinear figures may be separated into junctions, terminations, and curve segments by labeling every pixel  $p$  with a measure of local branching, and then thresholding the result. This measure is based on the *exit count*,  $x_{\square}$ , of each square region  $R$  of width  $w$  that contains  $p$ . The exit count of  $R$  is the number of times that black components intersect the border of  $R$ . If  $R$  has suitable size and position, the exit count reflects the local branching of a curvilinear figure that  $R$  partially includes. E.g., the exit count is four if  $R$  just contains a crossing of two lines. (Wojcik [21] applied this measure in a serial context.)

To allow varying curve width, given constant  $w$ , the curves are first thinned by a distance transform-based method. The positioning issue is addressed using a voting scheme—heuristically, meaningful positionings are more common than spurious ones. Given exit count at every position, the branching factor at a pixel  $p$  is taken to be the modal value of exit count among regions that include  $p$ . The exit and vote counts are computed hierarchically.

**Counting border edges.** The border,  $b$ , of  $R$  is the set of pixels in  $R$  with a four-neighbor not in  $R$ . The border edge count,  $e_{\square}$ , is the number of black-white transitions between pixels in  $b$ . It is computed hierarchically by combining corresponding partial counts for the left, right, top, and bottom borders of  $R$ 's children, introducing border edges where adjacent corner pixels of the children have different colors.

The exit count is half the border edge count:  $x_{\square} = 1/2e_{\square}$ .

Let  $c_{tl}, c_{bl}, c_{tr}, c_{br}$  denote one-bit values representing the top-left, bottom-left, top-right, bottom-right corner pixel, respectively, of  $r$ . For  $l = -1$ ,  $c_{tl} = c_{bl} = c_{tr} = c_{br} = \mathbf{i}$ . For  $l > -1$ ,

$$c_{tl} : \mathcal{P} \leftarrow \mathcal{C}_1 \quad (14)$$

$$c_{bl} : \mathcal{P} \leftarrow \lambda(\mathcal{C}_1 \vee \mathcal{C}_2^w, \mathcal{C}_1) \quad (15)$$

$$c_{tr} : \mathcal{P} \leftarrow \lambda(\mathcal{C}_1, \mathcal{C}_1 \vee \mathcal{C}_2^w) \quad (16)$$

$$c_{br} : \mathcal{P} \leftarrow \mathcal{C}_1 \vee \mathcal{C}_2^w \quad (17)$$

Let  $e_l, e_r, e_t, e_b$  denote the left, right, top, bottom partial border edge counts. For  $l = -1$ ,  $e_l = e_r = e_t = e_b = e_{\square} = -1$ . For  $l > -1$ ,

$$e_l : \mathcal{P} \leftarrow \lambda(\mathcal{C}_1 + \mathcal{C}_2 + \delta_l, \mathcal{C}_1) \quad (18)$$

where  $\delta_l = \lceil \mathcal{C}_1(c_{bl}) \neq \mathcal{C}_2^b(c_{tl}, c_{bl}) \rceil$ .

$$e_t : \mathcal{P} \leftarrow \lambda(\mathcal{C}_1, \mathcal{C}_1 + \mathcal{C}_2 + \delta_t) \quad (19)$$

where  $\delta_t = \lceil \mathcal{C}_1(c_{tr}) \neq \mathcal{C}_2^b(c_{tl}, c_{tr}) \rceil$ .

$$e_r = \mathcal{T}(e_l, \alpha_x(l+1) - 1, 0) \quad (20)$$

$$e_b = \mathcal{T}(e_t, 0, \alpha_y(l+1) - 1) \quad (21)$$

$$e_{\square} : \mathcal{P} \leftarrow \gamma(v, e_l + e_r + e_t + e_b, \mathcal{C}_1) \quad (22)$$

**Voting for branching factor.** Consider the set  $S$  of  $w \times w$  regions that include a particular pixel  $p$ . Consider the region  $R$  in  $S$  for which  $p$  is the lower right corner pixel. The upper left corner pixel of every region in  $S$  is in  $R$ . Therefore, to count the votes with respect to every pixel, for a given value  $v$  of the exit count, we count the instances of  $v$  within  $w \times w$  regions—i.e., summing hierarchically up to level  $2 \log w$ —and then translate the result downward and rightward by  $w$ . I.e., let  $s_{\square}$  denote the votes for exit count  $v$ . For  $l = -1$ ,  $a_{\oplus} = \lceil x_{\square} = v \rceil$ . For  $l > -1$ ,  $a_{\oplus} : \mathcal{P} \leftarrow \mathcal{C}_1 + \mathcal{C}_2$ . Then  $s_{\square} = \mathcal{T}(a_{\oplus}, -w, -w)$ .

Branching factor is the modal value of  $x_{\square}$  over its meaningful range. In practice, all values of  $x_{\square}$  greater than 4—which indicate *multiway* branches—are mapped to the value 5, and  $s_{\square}$  is computed in the range 0 to 5. *Terminations, segments, forks, and crossings* are associated with branching factors 1, 2, 3, and 4, respectively.

## 6 Near neighbor linking

This section presents a scheme for computing the distance transform<sup>2</sup> [4] and an operation for communicating between near neighbors. These operations have widespread applications to scene analysis [16] [14].

<sup>2</sup>The distance transform assigns to each pixel the distance to the nearest non-zero pixel.

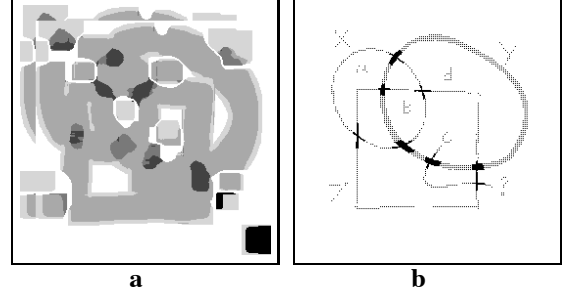


Figure 7: (a) Grey value at a pixel represents the branching factor of an associated  $w \times w$  region. (b) Junction pixels, with branching factor of 3 or 4, are shown in black.

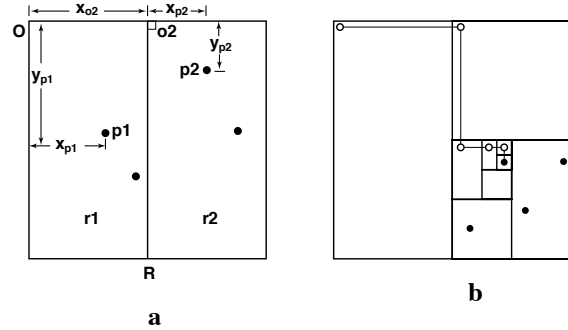


Figure 8: (a) The nearest neighbor of  $O$  in  $R$  is computed hierarchically from results for  $r1$ ,  $r2$ , and the offset of  $o2$  from  $O$ . (b) One bit per node represents hierarchical paths linking each black pixel to every pixel of which it is a near neighbor.

**Hierarchical quadrant near neighbors.** Consider a rectangular region  $R$  with upper-left corner pixel  $O$  (Figure 8).  $O$  is termed the *origin* of  $R$  with respect to the fourth quadrant. We can estimate hierarchically the black pixel in  $R$  nearest to  $O$ . Divide  $R$  into two rectangular subregions  $r1$  and  $r2$ , with respective origins  $O$  and  $o2$ , where the offsets of  $o2$  from  $O$  are  $x_{o2}, y_{o2}$ . Suppose we know  $x_{p1}, y_{p1}$ , the offsets of the nearest black pixel  $p1$  to  $O$  in  $r1$ , and  $x_{p2}, y_{p2}$ , the offsets of the nearest black pixel  $p2$  to  $o2$  in  $r2$ . Then the offsets of  $p2$  from  $O$  are  $x_{o2} + x_{p2}, y_{o2} + y_{p2}$ , so we can decide which of  $p1$  and  $p2$  is nearer to  $O$ .

$R$  and  $O$  may be defined with respect to any quadrant. E.g., for the first quadrant,  $O$  is the lower-left corner pixel of  $R$ . The signs of the second child offsets are given in Table 1 (a) for each quadrant.

Let  $i = \gamma(-\mathbf{i}, \infty, 0)$ . Let  $x_R, y_R, d_R$  denote the  $x$  offsets,  $y$  offsets, distances of the nearest black pixel to  $O$  in  $R$ , with respect to quadrant  $q$ . Let  $x_1, y_1, d_1$  denote the  $x$  offsets,  $y$  offsets, distances of the nearest black pixel to  $O$  in  $r1$ . For  $l = -1$ ,  $x_1 = y_1 = d_1 = i$ . For  $l > -1$ ,

$$x_1, y_1, d_1 : \mathcal{P} \leftarrow \mathcal{C}_1 \quad (23)$$

5 Let  $x_2, y_2, d_2$  denote the  $x$  offsets,  $y$  offsets, distances of the

$q$	$s_x^q(q)$	$s_y^q(q)$
1	1	-1
2	-1	-1
3	-1	1
4	1	1

a

$q$	$o_x^q(q)$	$o_y^q(q)$
1	1	0
2	0	-1
3	-1	0
4	0	-1

b

Table 1: (a) Signs for the second child offsets with respect to quadrant. (b) Offsets for the skewing operation.

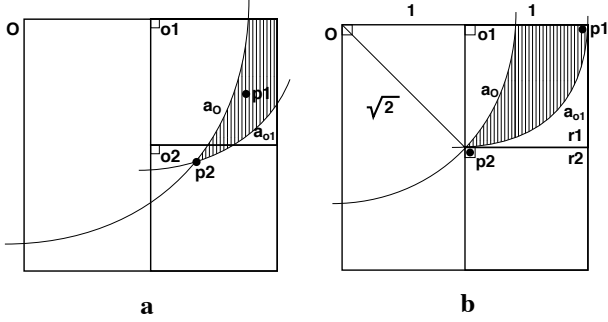


Figure 9: Hierarchical neighbor errors under Euclidean distance (shading regions for  $p2$  are shaded): (a)  $p1$  shadows  $p2$  with respect to  $O$ . (b) Worst case.

nearest black pixel to  $o_2$  in  $r_2$ .

$$x_2 : \mathcal{P} \leftarrow \mathcal{C}_2 + o_x(l) o_x^q(q) \quad (24)$$

$$y_2 : \mathcal{P} \leftarrow \mathcal{C}_2 + o_y(l) o_y^q(q) \quad (25)$$

$$d_2 = (x_2^2 + y_2^2)^{1/2} \quad (26)$$

Let  $\pi = \lceil d_2 < d_1 \rceil$ . Now we select the first and second child results:  $x_R = \gamma(\pi, x_2, x_1)$ , and similarly for  $y_R, d_R$ .

**Error analysis.** Using Manhattan distance, this method gives the Manhattan nearest neighbor. Using Euclidean distance, however, it does not necessarily give the Euclidean nearest neighbor, depending on the spatial arrangement of the data points, as Figure 9 (a) illustrates.  $r_2$  is divided into two square subregions, with origins  $o_1$  and  $o_2$ . First,  $p_1$  is correctly determined to be nearer to  $o_1$  than  $p_2$ . Then,  $p_1$  is incorrectly determined to be closer to  $O$  than  $p_2$ —we say  $p_1$ , or any black pixel in the shaded region, *shadows*  $p_2$ . Figure 9 (b) shows the worst case construction. The neighbor found can be up to  $\sqrt{2}$  further away than the Euclidean nearest neighbor. (Also, the neighbor found and the Euclidean neighbor may be up to  $\sqrt{2}$  apart.) By this construction, however, the shadowing region is at most  $1/8$ th the area of  $R$ , which for many applications gives a tolerable probability of error. The shadowing region is reduced to  $1/32$ nd of  $R$  if the offsets of  $p_1, p_2$  from  $r_1, r_2$ , respectively, are both passed upwards for the near neighbor decision.

Figure 10 compares a Voronoi tessellation obtained using our method to one based on exact Euclidean distance.

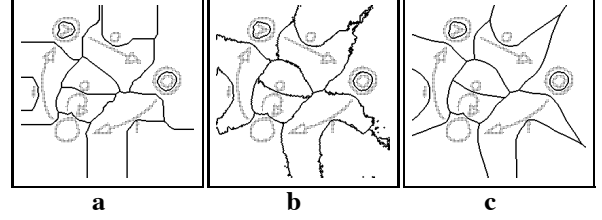


Figure 10: Voronoi boundaries based on: (a) our hierarchical DT, Manhattan distance; (b) our hierarchical DT, Euclidean distance; (c) a sequential Euclidean DT.

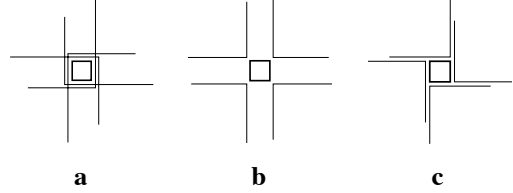


Figure 11: Three ways of defining the quadrants in a pixel array: (a) overlapping; (b) non-adjacent; (c) skewed.

**Skewing.** Because  $R$  includes  $O$ , the result of the above method is *self-inclusive*:  $x, y$  are both 0 at a black pixel, i.e.,  $O$  is treated as its own nearest neighbor. By defining the quadrants so that  $O$  is not in  $R$ , as in Figure 11 (c), we can get *self-exclusive* results at black pixels. This involves a simple adjustment, called *skewing*, of the self-inclusive results. Skewing makes the central pixel,  $c$ , in Figure 11 (c) the origin, and establishes near neighbor offsets for  $c$  based on those of  $O$  and on the offsets of  $O$  from  $c$ . I.e., if  $O$  is black, it is  $c$ 's near neighbor; if  $O$  is white,  $c$ 's neighbor is  $O$ 's neighbor, but the associated offsets and distance must be adjusted appropriately. The offsets of  $O$  from  $c$  in quadrant  $q$ , denoted by  $o_x^q(q), o_y^q(q)$ , are shown in Table 1(b).

Let  $x, y, d$  denote the self-exclusive results.

$$x = \begin{cases} o_x^q(q) & \text{if } \mathcal{T}(\mathbf{i}, o_x^q(q), o_y^q(q)) \\ x' & \text{if } d_R > 0 \\ 0 & \text{otherwise} \end{cases} \quad (27)$$

where  $x' = \mathcal{T}(x_R, o_x^q(q), o_y^q(q)) + o_x^q(q)$ .

$$y = \begin{cases} o_y^q(q) & \text{if } \mathcal{T}(\mathbf{i}, o_x^q(q), o_y^q(q)) \\ y' & \text{if } d_R > 0 \\ 0 & \text{otherwise} \end{cases} \quad (28)$$

where  $y' = \mathcal{T}(y_R, o_x^q(q), o_y^q(q)) + o_y^q(q)$ . Then,  $d = (x^2 + y^2)^{1/2}$ .

**Linking and Reading.** A pixel  $o$  is called an *owner* of a pixel  $p$  if  $p$  is a near neighbor of  $o$ . To support an efficient operation, called *Read*, for transmitting data from neighbors to owners, we extend the above algorithm so it stores communication paths between them (see Figure 8 (b)). The extended process is called *Linking*. The paths are represented

in a distributed, hierarchical fashion, using one bit per level per pixel. This *link* bit,  $\pi$ , records whether  $r1$  or  $r2$  contained the neighbor of  $O$ :  $\pi = [d_2 < d_1]$ . In an in-place implementation,  $\pi$  is preserved by concatenation into a bit string as it is computed at each level.

The Read operation involves two steps. The first is a hierarchical process that uses  $\pi$  to transmit data from child to parent:  $\mathcal{R}_i : \mathcal{P} \leftarrow \gamma(\pi, \mathcal{C}_2, \mathcal{C}_1)$ . The second step skews the results:  $\mathcal{R} = \mathcal{T}(\mathcal{R}_i, o_x^q(q), o_y^q(q))$ .

**Distance and neighbor value transforms.** The distance transform,  $d_m$ , is given by minimizing  $d$  over all quadrants. To read from the nearest neighbor, we must Read with respect to each quadrant, and then at each location select the result for the quadrant associated with  $d_m$ .

## 7 Related work

We discuss three areas of related work: previous extensions to hierarchies that reduce position dependence; a previous application of exhaustive hierarchy; and non-hierarchical parallel spatial integration schemes.

**Hierarchy extensions.** In *dynamically linked* pyramids [6], parent-child links are set up and changed during a computation. E.g., a node at level  $l$  may establish child links to any four nodes within a  $4 \times 4$  block of nodes at level  $l - 1$ . *Overlapped pyramids* [6] [8] are the superposition of two pyramids,  $a$  and  $b$ , such that base nodes are shared and corresponding non-base nodes of  $a$  and  $b$  represent regions that overlap, e.g., by 50%. Neither extension gives position invariance. Exhaustive hierarchy takes overlap to the extreme.

Shafir [19] gives a fascinating exploration of the effects of node fan-in, degree of region overlap, and region shape on the speed of hierarchical connected component labeling. His labeling processes involve networks with as few as two hierarchical levels, fan-in of up to  $N$ , and overlap ranging up to the maximum. He did not apply these generalized hierarchical networks to other spatial integration problems.

**Intrinsic parallelism.** Frederickson and McBryan [9] describe multiscale methods that, like ours, utilize the same number of processors at every level and emerged from mapping conventional hierarchical algorithms to a fine-grained hypercube architecture. Theirs are multigrid methods for solving PDE systems. They call their methods “intrinsically parallel” because, like ours, they fully exploit the parallelism of fine-grained machines, but would be prohibitively costly on a serial or multi-processor machine. They did not apply the approach to geometric analysis.

**Non-hierarchical methods.** *Distance transform:* Yamada [23] describes an iterative mesh algorithm that computes the Euclidean distance transform in  $O(N)$  time. Distance information is radially propagated from figure boundaries by re-

peated parallel updates using a fixed, small (e.g.,  $3 \times 3$ ) local neighborhood operation. Another class of methods propagates one-dimensional distance information, say along pixel rows, and then combines the results into a two-dimensional transform [16], [5]. These methods seem subject to  $O(N)$  time mesh and  $O(\log N)$  time hypercube implementations, but no such implementations have been published. (Divide-and-conquer Voronoi diagram algorithms, which are somewhat related, take polylog and  $O(N)$  time on the hypercube and mesh respectively [10] [1].)  $O(N)$  time on the mesh and  $O(\log N)$  time on the hypercube are optimal, since the input image may contain just two pixels at opposite corners of the array; these are the optimal one-to-one routing times in each case [12]. Our pointwise hierarchical distance transform is  $O(\log N)$  on the hypercube and  $O(N)$  on the mesh, but its results are not exact. (Samet [18] presents a hierarchical chessboard distance transform for quadtrees, but it doesn’t give pointwise results.)

*Feature detection:* The local features of Section 5 and the local properties of Section 4 were expressed as functions of certain primitive measurements computed by summing over local regions. These local sums may be computed at every position based on *initial prefix* operations—i.e., cumulative summing—along pixel rows and columns. The sum in an interval  $i$  of a pixel row/column is the difference between the cumulative sums at the ends of  $i$ . The sum in a region is given by summing the row interval sums within vertical intervals, or vice versa [22], [14]. The sums at every location for an arbitrary but fixed region size are given in  $O(N)$  time on a mesh, using initial prefix operations that are iterative within rows/columns but parallel across them. They are given in  $O(\log N)$  time on a hypercube, using the *parallel prefix* operation [3]. Our feature detection methods are  $O(\log N)$  on the hypercube and  $O(N)$  on the mesh.

*Connected component labeling:* Labeling connected components on a mesh by straightforward iterative propagation of labels among adjacent pixels takes  $O(N^2)$  time, due to the unrealistic worst case of a space-filling curve. The run time is proportional to the *diameter* of the largest component—the maximum of all minimal path lengths between two points in the component—which is typically much closer to  $N$  than  $N^2$ . Levaldi’s [13] component labeling method, which also involves iterative propagation, runs in  $O(N)$  time due to a clever propagation scheme, but requires  $O(N)$  bits of storage per pixel. Cypher et. al. [7] present a  $O(\log N)$  time EREW PRAM divide-and-conquer component labeling scheme, based on reduction operations on linked lists. It runs in  $O(\log^2 N)$  time on a hypercube [20], and in  $O(N)$  time on a mesh. (The mesh implementation uses a  $O(N)$  linked-list reduction algorithm given in [2].) This approach is attractive in its insensitivity to shape and configuration, but it is algorithmically complex—it uses both sorting and packet routing as subroutines.

	Connectivity	Distance	Features
pyramid	$O(N)$	none	none
mesh (NH)	$O(N)$	$O(N)$	$O(N)$
mesh (EH)	$O(mN)$	$O(N)$	$O(N)$
h-cube (NH)	$O(\log^2 N)$	none	$O(\log N)$
h-cube (EH)	$O(m \log N)$	$O(\log N)$	$O(\log N)$

Table 2: Complexity of position-invariant spatial integration for an  $N \times N$  image, for exhaustive hierarchical (EH) vs. non-hierarchical (NH) methods.  $m$  is  $N^2$  in the worst case, but closer to  $\log N$  or  $N$  for realistic inputs.

Our global component labeling method is  $O(m \log N)$  on the hypercube and  $O(mN)$  on the mesh, where  $m$  varies with shape and configuration of the components and is  $N^2$  in the worst case. Empirical characterization of the performance of this method remains for future work; in our experience,  $m$  is similar to  $\log N$  in typical cases and to  $N$  in the worst realistic cases. The method is attractive in its simplicity and its highly regular pattern of communication.

## 8 Conclusion

This paper endowed simple hierarchical processes with position invariance by extending the tree structure into an exhaustive hierarchy. By presenting detailed algorithms for several key spatial integration processes, we demonstrated that exhaustive methods transcend the main limitations of tree and pyramid approaches, and do so with the minimum of algorithmic complexity, greater scope, and excellent time and space performance. All algorithms were precisely and completely specified in an extremely compact form, demonstrating their simplicity. Most are asymptotically optimal for the mesh and hypercube, with very small constant factors, and none involve global routing by unique addresses. They all involve constant space per hierarchical level; each level of an upward process can be pipelined to give results in constant time. Also, exhaustive architecture enabled, for the first time, a straightforward application of hierarchy to the pointwise distance transform.

## References

- [1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Proc. Symp. Found. Comp. Sci.*, 1985: 468–477.
- [2] M. Atallah and S. Hambrusch. Solving tree problems on a mesh-connected processor array. *Information and Computation*, 69(1-3):168–187.
- [3] G. Blelloch. Scans as primitive parallel operations. *Proc. Int. Conf. on Parallel Processing*, 1987: 370–376.
- [4] G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing* 34, 1986:321–345.
- [5] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman. Linear time Euclidean distance transform algorithms. *IEEE Trans. on Pattern Anal. and Machine Intel.*, 17, 5, 1995:529–533.
- [6] P. Burt, T. Hong, and A. Rosenfeld. Segmentation and estimation of image region properties through cooperative hierarchical computation. Computer Science Technical Report TR-927, Univ. of Maryland, 1980.
- [7] R. Cypher, J. Sanz, and L. Snyder. EREW PRAM and Mesh Connected computer algorithms for image component labeling. *IEEE Trans. Pat. Anal. and Machine Intel.*, 11, 1989:258–262.
- [8] C. R. Dyer. Pyramid algorithms and machines. In *Multicomputers and image processing*, ed. K. Preston, Jr and L. Uhr, 1982:409–412.
- [9] P. Frederickson and O.A. McBryan. Intrinsically parallel multiscale algorithms for hypercubes. *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications, Volume II—Applications*, Pasadena, CA, 1988: 1726–1734.
- [10] C. Jeong and D. Lee. Parallel geometric algorithms on a mesh computer. Northwestern Univ. TR 87-02-FC-01, 1987.
- [11] V. Prasanna-Kumar and M. Eshaghian. Parallel geometric algorithms for digitized pictures on mesh of trees. *Proc. 1986 Int. Conf. on Parallel Proc.*, 1986:270–273.
- [12] T. Leighton. *Introduction to parallel architectures: arrays, trees, hypercubes*. Cambridge, MA: The M.I.T. Press, 1992.
- [13] S. Levialdi. On shrinking binary picture patterns. *Communications of the ACM*, 15 1, 1972: 7–10.
- [14] J. Mahoney. Signal-based figure/ground separation. Submitted to *IEEE Int. Symp. on Computer Vision*, 1995.
- [15] R. Miller, V. Prasanna-Kumar, D. Reisis, and Q. Stout. Image processing on reconfigurable meshes. *From Pixels to Features II* H. Burkhardt, Y. Neuvo, J. C. Simon (eds.), North-Holland, Amsterdam, 1991:85–101.
- [16] D. Paglieroni. Distance transforms: properties and machine vision applications. *Computer Vision, Graphics, and Image Processing* 53 (1), 1982: 56–74.
- [17] T. Pavlidis. *Algorithms for graphics and image processing*. Rockville, MD, 1982: Computer Science Press.
- [18] H. Samet. Distance transform for images represented by quadtrees. *IEEE Trans. on Pattern Anal. and Machine Intel.* 4 (3), 1982:298–303.
- [19] A. Shafir. Fast region coloring and the computation of inside/outside relations. M.Sc. Thesis. Rehovot, Israel: Dept. of Applied Math., Weizmann Institute of Science, 1985.
- [20] Q. Stout. Mapping vision algorithms to parallel architectures. *Proc. of the IEEE* 76 (8), 1988:982–995.
- [21] Z. Wojcik. An approach to the recognition of contours and line-shaped objects. *Computer Vision, Graphics, and Image Processing*, 25, 1984: 184–204.
- [22] J. Woodfill. Motion Vision and Tracking for Robots in Dynamic, Unstructured Environments. Report No. STAN-CS-92-1440. Stanford, CA, 1992: Stanford University.
- [23] H. Yamada. Complete Euclidean distance transformation by parallel operation. *Proc. 7th Int. Conf. on Pattern Recognition*, Montreal, Canada, 1984:69-71.