COMPUTER VISION AND IMAGE UNDERSTANDING
Vol. 62, No. 1, July, pp. 1–10, 1995

Hypercube Algorithms for Parallel Processing of Pointer-Based Quadtrees

FRANK DEHNE* AND ANDREW RAU-CHAPLIN†

School of Computer Science, Carleton University, Ottawa K1S 5B6, Canada

AND

AFONSO G. FERREIRA‡

Laboratoire de l'Informatique du Parallelisme - CNRS, Ecole Norm. Sup. de Lyon, 69364 Lyon, Cedex 07, France

Received October 9, 1990; accepted June 13, 1994

This paper studies the parallel construction and manipulation of pointer-based quadtrees on fine grained hypercube multiprocessors. Previous papers considered the parallel processing of *linear* quadtrees. Here we show that parallel *pointer-based* quadtrees are a viable alternative. We first solve the problem of efficiently constructing a pointer-based (or linear) quadtree from an image represented either by a binary matrix or a boundary code. Then we present efficient parallel manipulation algorithms for pointer-based quadtrees, such as finding the neighbors of all leaves in a quadtree or computing the union/intersection of two quadtrees. These algorithms improve on existing time complexities and can be implemented in fine grained hypercube systems (e.g., the Connection Machine CM2). In the expected case, the space complexity is the same as for previous methods. In the worst case (of a degenerated quadtree), the space complexity increases by a factor which, for the hypercube, is smaller than the time complexity improvement. As a byproduct of our hypercube algorithms, we also obtain some PRAM algorithms for quadtrees that improve on known results. © 1995 Academic Press, Inc.

1. INTRODUCTION

A *region quadtree* is a well-known hierarchical data structure for representing a binary image of size $\sqrt{M} \times \sqrt{M}$ ($\sqrt{M} = 2^r$ for some positive integer r). The root of the quadtree represents the entire image and has a value "black," "white," or "gray" depending on whether the entire image is black, white, or composed of both types of pixels, respectively. If the root is gray, it has four children which are the roots of quadtrees recursively representing the four quadrants of the image; otherwise it has no children. For the remainder, we do not differentiate between a node of a quadtree and the portion of the image represented by that node.

There are two widely used representations of quadtrees. A *pointer-based quadtree* uses the standard tree representation. A *linear quadtree* can be either a preorder traversal of the nodes of a quadtree or the sorted sequence (with respect to the preorder of the tree) of the quadtree's leaves. Some linear quadtree representations of the second type store with each leaf also a code sequence representing the path from the root to that leaf (*linear quadtree with path encoding*) while others store for each leaf only its size and location (*linear quadtree without path encoding*). For an overview and bibliography on quadtrees and applications we refer to the work of Samet ([18]).

Quadtrees are a very useful and widely used data structure for image processing, and quadtree algorithms for a number of image-processing tasks have been developed [18]. Recently, researchers have also started to consider quadtree algorithms for parallel models of computation [2, 7, 9, 12, 13]. While some papers [12, 13] consider parallel architectures designed (or reconfigured) particularly for quadtree manipulation, others [2, 9] consider the general purpose architectures mesh-connected computer and PRAM, respectively. Hung and Rosenfeld [9] study mesh-connected computer algorithms for constructing and manipulating linear quadtrees without path encoding, while PRAM algorithms for manipulating linear quadtrees with path encoding are studied by Bhaskar *et al.* [2].

Table 1 lists the parameters that will be used for the remainder of this paper. Best results on the mesh are listed in Tables 2 and 3 (rightmost column). PRAM algo-

* Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

† Research partially supported by the Bell-Northern Research Graduate Award Program.

‡ Part of this work was done while on leave from the University of Sao Paulo (Brazil), project BID/USP. Research partially supported by CAPES/COFECUB (Grant 503/86-9). Support from the French C3 is acknowledged.

1077-3142/95 $12.00
Copyright © 1995 by Academic Press, Inc.
All rights of reproduction in any form reserved.

TABLE 1
Overview of Parameters

M	Number of pixels in the original image
b	Length of the boundary code
N	Size of the *pointer-based quadtree*
N'	Size of the *linear quadtree with path encoding*
n	Size of the *linear quadtree without path encoding*
h	Height of the quadtree
t	Time complexity
s	Total memory space
p	Number of processors

rithms for manipulating linear quadtrees are listed in Table 3 (second column from the right). The hypercube time and space complexities listed in Table 3 for manipulating linear quadtrees with path encoding are obtained from [2] by using standard PRAM simulation on a hypercube, as described by Nassimi and Sahni [14], together with Cypher and Plaxton's deterministic hypercube sorting algorithm [4].

In this paper, we study parallel construction and manipulation of quadtrees on fine grained hypercube multiprocessors (hypercubes with a large number—more than 10,000—of small processors). The Connection Machine CM2 is an example of an existing fine grained system. We extend previous results for the hypercube in two directions.

(1) We study parallel quadtree construction algorithms for the hypercube. We describe algorithms for converting images, represented by either a binary array or a boundary code, into pointer-based as well as linear quadtrees. Table 2 summarizes these results.

(2) We study parallel hypercube methods for manipulating pointer-based quadtrees. We show that, for parallel processing, pointer-based quadtrees are an attractive alternative to the linear quadtrees studied in the previous literature. The parallel manipulation algorithms for pointer-based quadtrees presented in this paper improve, in the *expected case*, on previously presented methods. Table 3 summarizes our contribution. Note that the algorithms in [2] apply to linear quadtrees *with* path encoding. In the expected case, the height, h, of the quadtree is $O(\log N)$ ([1, 8, 10]. Hence, $N = O(N')$; i.e., the linear and pointer-based quadtrees have, asymptotically, the same space requirement. In this case, we obtain improvements in the time complexity for several problems, such as computing the neighbors of all leaf nodes and the perimeter of an image [$O(h \log N)$ vs $O(h \log^2 N \log^2 \log N)$] or computing the union/intersection of two quadtrees [$O(h \log N)$ vs $O(h \log^2 N \log^2 \log N)$]. In the *worst case*, $h = O(N)$, the linear quadtree with path encoding needs to store one path requiring $O(h)$ bits, while the pointer-based quadtree needs $O(h)$ pointers of $O(\log h)$ bits each; that is, $N = O(N' \log h)$. Then, we obtain a time space trade-off between the above time complexity improvements and increased storage for pointer-based quadtree algorithms. Note that the space increases by a factor smaller than the time complexity improvement.

As a byproduct of our hypercube algorithms, we also obtain some PRAM algorithms for quadtrees (see Table 3). Note that the emphasis of this paper is on hypercube algorithms, and these straightforward observations for the PRAM are not necessarily optimal. They are listed only because either no previous PRAM algorithm existed or they improve on existing results.

The remainder of this paper is organized as follows. In Section 2, we discuss some preliminaries concerning the models of parallel computation and describe a technique called the *dynamic multiway search paradigm*. In Section 3, we present efficient hypercube algorithms for *constructing* a (pointer-based or linear) quadtree from a binary image or from an image represented by its boundary code. In Section 4, we introduce efficient parallel hypercube algorithms for manipulating pointer-based quadtrees. Section 5 contains some observations for the PRAM, and Section 6 concludes the paper.

TABLE 2
Parallel Quadtree Construction Methods (New Results Are in Boldface)

	Pointer-based quadtree		Linear quadtree		
Problem	**Hypercube**	PRAM	**Hypercube**	PRAM	Mesh
Convert image to quadtree	$\mathbf{s = p = M}$, $\mathbf{t = O(\log^2 M)}$	$s = M$, $p = M/\log M$, $t = O(\log M)$	$\mathbf{s = p = M}$, $\mathbf{t = O(\log^2 M)}$[a]	$s = M$, $p = M/\log M$, $t = O(\log M)$	$s = p = M$, $t = O(\sqrt{M})$ [9]
Convert boundary code to quadtree	$\mathbf{s = p = b}$, $\mathbf{t = O(h \log b)}$	$s = p = b$, $t = O(h \log b)$[a]	$\mathbf{s = p = b}$, $\mathbf{t = O(h \log b)}$	$s = p = b$, $t = O(h \log b)$[a]	

[a] Simple consequence of the respective hypercube result.

TABLE 3
Parallel Quadtree Manipulation Methods (New Results Are in Boldface)

Problem	Pointer-based quadtree		Linear quadtree		
	Hypercube	PRAM	Hypercube	PRAM	Mesh
Determine neighbors of all leaf nodes/compute perimeter	$\mathbf{s = p = N}$, $\mathbf{t = O(h \log N)}$	$s = p = N$, $t = O(h)^a$	$s = p = N'$, $t = O(h \log^2 N' \log^2 \log N')$ $s = p = O(4^h) \geq O(N)$, $t = O(h \log N' \log^2 \log N' + \log^2 N' \log^2 \log N')^b$	$s = p = N'$, $t = O(h \log N')$ $s = p\ O(4^h) \geq O(N)$, $t = O(h + \log N')$ [2]	$s = p = n$, $t = O(\sqrt{n})$ [9]
compute area/centroid	$s = N$ $p = N$, $t = O(\log N)^c$	$s = N$, $p = N/\log N$, $t = O(\log N)^c$	$s = p = N'$, $t = O(\log N')^c$	$s = N'$ $p = N'$ $t = O(\log N')$ [2]	$s = p = n$, $t = O(\sqrt{n})$ [9]
Rotate by i*90°	$\mathbf{s = p = N}$, $\mathbf{t = O(h + \log N)}$	$s = p = N$, $t = O(h)^a$			
Compute union intersection	$\mathbf{s = p = N}$, $\mathbf{t = O(h \log N)}$	$s = N$, $p = N/\log N$, $t = O(\log N)$ [11]	$s = p = N'$, $t = O(h \log^2 N') \log^2 \log N')^b$	$s = p = N'$, $t = O(h \log N')$ [2]	$s = p = n$, $t = O(\sqrt{n})$ [9]
Compute complem.	$s = p = N$, $t = O(1)^{c,d}$	$s = p = N$, $t = O(1)^c$	$s = p = N'$, $t = O(h \log N' \log^2 \log N')^b$	$s = p = N'$, $t = O(h)$ [2]	$s = p = n$, $t = O(\sqrt{n})$ [9]
Determine connected comp.					$s = p = n$, $t = O(\sqrt{n})$ [9]

[a] Simple consequence of the respective hypercube result.
[b] Follows from [2] by standard PRAM simulation on a hypercube as described in [12], together with [4].
[c] Obvious, and listed for completeness only.
[d] Assumes $O(1)$ time host to nodes instruction broadcast (e.g., connection machine).

2. PRELIMINARIES

Before presenting our quadtree algorithms, we introduce some notations and previous results which will be used in the remainder. We start by defining the parallel models of computation used in this paper.

2.1. *The Hypercube Multiprocessor*

A hypercube multiprocessor is a set $P_1, \ldots, P_p$ of p processors connected in a hypercube fashion; i.e., P_i and P_j are connected by a communication link if and only if the binary representations of i and j differ in exactly 1 bit. In a hypercube, there is no shared memory. The entire storage capability consists of constant size local memories, one attached to each processor ($s = O(p)$).

2.2. *Storing Pointer-Based Quadtrees on a Hypercube Multiprocessor*

For the hypercube, we require a scheme for distributing a quadtree over its local memories. Consider the *level order numbering* of the nodes of a quadtree as indicated in Fig. 1. For the remainder we will assume that each node with level order number i, together with the attached data and pointers to its children, is stored at processor P_i.

In most of the algorithms presented henceforth, we will use the *shuffled row-major* (SRM) numbering of a binary image represented in a grid G. Contrarily to the *row-major* (RM) numbering, where pixels are numbered row after row, from left to right (as shown in Fig. 2a), the SRM numbering is obtained from the RM numbering as follows. Let i be the RM number of position $G(j, k)$, and

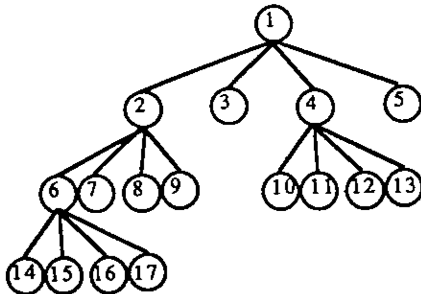


FIG. 1. Level order numbering of the nodes of a quadtree.

let $b_1 b_2 \cdots b_q$ be the binary representation of $(i - 1)$. Further, let $b_1 b_{(q/2)+1} b_2 b_{(q/2)+2} \cdots b_{q/2} b_q$ (the shuffling of $b_1 b_2 \cdots b_q$) represent the integer i'. Then i' is the SRM number of $G(j, k)$. See Fig. 2b.

On the hypercube, the conversion between SRM and RM numbering can be performed in time $O(\log N)$ by a bit-permute-complement operation as described in [15], provided that the SRM/RM number and hypercube address are identical.

2.3. *Multiway Search on a Tree*

Let $T = (V, E)$ be a tree of size k, height h, and outdegree $O(1)$, and let U be a universe of possible search queries on T. A *search path* for a query $q \in U$ is a sequence $path(q) = (v_1, \ldots, v_h)$ of h vertices of T defined by a *successor* function f: $(V \cup \{start\}) \times U \Rightarrow V$. That is, f is a function with the property that $f(start, q) \in V$ and for every vertex $v \in V$, $(v, f(v, q)) \in E$ or $f(v, q), v) \in E$. A *search process* for a query q with search path $(v_1, \ldots, v_h)$ is a process divided into h time steps $t_1 < t_2 < \cdots < t_h$ such that at time t_i, $1 \le i \le h$, there exists a processor which contains (in its local memory) a description of both the query q and the node v_i. Note that we do not assume that the search path is given in advance. We assume that it is constructed "online" during the search by successive applications of the function f. Given a set $Q = \{q_1, \ldots, q_m\} \subseteq U$ of m queries, $m = O(k)$, then the *multiway search problem* consists of executing (in parallel) all m search processes induced by the m queries.[1]

The best way to visualize this process is to depict each search process as a pebble, representing the respective query and moving through the tree T. A pebble may only move along edges of T, but it can traverse them in both directions. The multiway search problem consists of m such pebbles moving simultaneously through the tree. At each time step, every pebble decides which node to "visit" next, and then all pebbles are simultaneously moved. Note that, each node of the tree may be visited, at any time, by an arbitrary number of pebbles.

On a PRAM (of size $\max\{k, m\}$) multiway search can be easily implemented in time $O(h)$. Each query (pebble) is simply represented by one processor, navigating it through the tree, stored in the shared memory. The PRAM's concurrent read capability ensures that queries visiting the same node do not interfere.

For hypercube multiprocessors, it was shown in [6] that the multiway search problem can be solved in time $O(h \log(\max\{k, m\}))$ on a hypercube of size $\max\{k, m\}$. The algorithm applies to a class of graphs called *ordered h-level graphs* [see [6] for a precise definition) which includes the class of all trees with constant degree. The following outlines the global structure of the algorithm (applied to the special case of search trees): Initially, the tree is stored as indicated in Section 2.2. The m search queries are stored in arbitrary order (with each processor storing at most one query). The m search processes for the m queries $q_1, \ldots, q_m$ are executed simultaneously in h phases, each requiring time $O(\log(\max\{k, m\}))$. Each phase moves all queries one step ahead in their search paths. In each phase, the queries are permuted such that they are sorted with respect to the level order number of the respective node they want to visit next. Furthermore, a copy of the search tree is created and its nodes are permuted such that, at the end of each phase, each processor containing a query q_i also stores a copy of the node the query wants to visit next. See [6] for a full description of the algorithm.

Consider the problem of changing the tree T or the set Q of queries during the execution of a multiway search. That is, during the search (more precisely, at the end of each phase of the algorithm outlined above) leaves may be added to T, subtrees may be deleted from T, and queries may duplicate or delete themselves. This problem is referred to as the *dynamic multiway search problem*. In [5] it has been shown that this problem can be solved, for the hypercube, such that the time complexity of each phase is still $O(\log(\max\{k, m\}))$. That is, the time complexity of the entire multiway search procedure for the dynamic case is still $O(h \log(\max\{k, m\}))$. For the

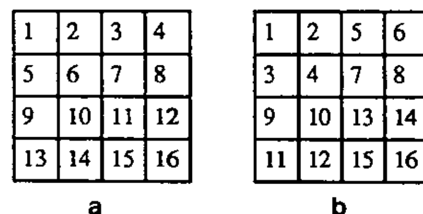


FIG. 2. (a) Row-major numbering. (b) Shuffled row-major numbering.

[1] In subsequent sections, queries will also be referred to as messages.

PRAM, the dynamic version also requires time $O(h \log(\max\{k, m\}))$. The problem here is that the assignment of processors to new queries and the assignment of storage space of deleted nodes to newly created ones may require a partial sum operation for each phase of the algorithm, which slows down the static solution by a factor of $O(\log(\max\{k, m\}))$.

3. HYPERCUBE ALGORITHMS FOR CONSTRUCTING QUADTREES FROM IMAGES AND BOUNDARY CODES

3.1. *Quadtree from Binary Image*

Consider a $\sqrt{M} \times \sqrt{M}$ binary image stored on a hypercube (with M processors) in row-major numbering. That is, processor P_i stores the pixel with row-major number i.

Computing a pointer-based quadtree from a binary image representation in $O(\log M)$ is immediate on the PRAM, with $s = M$ and $p = M/\log M$. Thus, simulation gives an upper bound of $O(\log^2 M (\log \log M)^2)$ for solving the same problem on a hypercube multiprocessor with $s = p = M$.

The following is an outline of an $O(\log^2 M)$ time parallel hypercube algorithm for computing a pointer-based quadtree from a binary image representation. (The implementation details will be presented afterward.)

(1) For each pixel (in parallel) its SRM number (as indicated in Section 2.2) is computed.

(2) All pixels are ordered according to the SRM numbering.

(3) A complete 4-ary tree, with the sorted sequence of pixels as leaves, is built.

(4) From each leaf a message is sent along the path to the root of the tree. The messages move synchronously upward from level to level. At each level, the following is executed:

If all four messages reaching a node x come from black {white} children, then x is set to black {white} and its children are marked "to be deleted." If the messages reaching x are from children with different color, x is set to gray.

(5) All nodes marked to be deleted are deleted, the remaining nodes are compressed to form a consecutive sequence, and all pointers are updated.

THEOREM 1. *The pointer-based quadtree representation of a $\sqrt{M} \times \sqrt{M}$ binary image can be computed in time $O(\log^2 M)$ on a hypercube with $s = p = M$.*

Proof. From the definition of quadtrees it follows that the tree generated by the above algorithm is the correct quadtree. What remains to be shown is that the above steps can be implemented within the claimed time complexity bounds. *Step* 1 requires only the local computation of the shuffled-row-major number of the respective pixel at each processor. For a $\sqrt{M} \times \sqrt{M}$ image, this takes $O(\log M)$ local computation steps. *Step* 2 requires time $O(\log M)$ because it reduces to the bit-permute-complement operation described in [15]. *Step* 3 can be implemented by building the tree level by level, starting with the leaves (which are given). Since it is a complete tree, at each stage the addresses of the nodes of the subsequent level can be immediately computed. This results in an $O(h \log M) = O(\log^2 M)$ time algorithm, because routing the nodes of the subsequent level to their respective positions reduces to a *concentrate* and *distribute* operation of [14]. *Step* 4 is a multiway search operation as outlined in Section 2.3, with traveling messages represented by query processes. Hence, it requires time $O(h \log M) = O(\log^2 M)$. Note that Step 4 does not change the topology of the tree but marks only the nodes to be deleted. In *Step* 5, the marked nodes are deleted by compressing the sequence of the remaining (nonmarked) nodes. This can be accomplished by a concentrate operation [14]. The problem of updating the pointers (address references between three nodes) can be solved by repermuting the tree to its original shape, communicating the new addresses between adjacent nodes through concentrate and distribute operations, and recompressing the tree. Hence, Step 5 requires time $O(\log M)$. ■

Linear quadtrees without path encoding can be constructed in essentially the same way by marking in Step 4 also gray nodes as to be deleted. For linear quadtrees with path encoding, we also need to compute (between Steps 4 and 5) the path encoding for each leaf by applying one additional multiway search procedure. Therefore, the linear quadtree representation (with or without path encoding) of a $\sqrt{M} \times \sqrt{M}$ binary image can also be computed in time $O(\log^2 M)$ on a hypercube with $s = p = M$.

3.2. *Quadtree from Boundary Code*

Consider an image I described by a *boundary code* of length b; i.e., a sequence $a_1, \ldots, a_b$ of b *boundary elements* $a_i \in \{r, l, u, d\}$ as shown in Fig. 3 (see [16]). The image I consists of the entire area inside the *boundary line* defined by the boundary code. The unit size pixels of I that are adjacent to the boundary line are called *boundary pixels* (see Fig. 3). For the remainder, let S_I denote a smallest (isothetic) square containing I. Note that S_I has a width of at most b.

Our hypercube algorithm for computing the pointer-based quadtree from the boundary code consists of two phases, each of which is outlined below.

Phase 1 computes a quadtree representing only the boundary pixels of I. We will refer to it as the *quadtree*

FIG. 3. Boundary code and boundary pixels of an image.

template. What remains to be done in Phase 2 is the creation of leaf nodes corresponding to the black and white area inside and outside the boundary line, respectively. The missing children of an internal node x, at the end of Phase 1, will be referred to as *absent children* of x Note that all absent children are leaves.

Phase 1. (1) For each boundary element, its absolute address (in S_I) is computed, and the adjacent boundary pixels are created (see [16]).

(2) The SRM number of each boundary pixel with respect to S_I is computed.

(3) All boundary pixels are ordered with respect to their SRM number.

(4) A quadtree with the above sequence of boundary pixels as leaves is built. For nodes with less than four children, for each missing child a pointer with value "absent" is created.

In order to build the final quadtree from the template created in Phase 1, we recall the following from [16].

LEMMA 1. [16] *After Phase* 1, *if an absent child of a node is black {white}, then all other absent children of a node are black {white}.*

LEMMA 2. [16] *After Phase* 1, *consider a node, x, with at least one absent child. Choose an absent child R adjacent to a nonabsent (black or gray) sibling Q, and a nonabsent leaf q in the subtree rooted at Q which is adjacent to R. If q is white then R is white. If q is black and adjacent to the boundary line, then R is white if the boundary line is between q and R, and black if the boundary line does not separate them. If q is black and not adjacent to the boundary line, then R is black.*

The following outlines the remainder of the algorithm.

Phase 2. (1) From each leaf, a message is sent to the root of the tree. The messages move synchronously upward from level to level. For each node, a value *Nodetype* is determined which indicates for each side of its respective quadrant whether it is completely inside the image I, completely outside of I, or intersected by the border line (see also [16]). Note that, the Nodetype value for every boundary pixel (leaf of template quadtree) is given; for every internal node, given the Nodetype values of all its children, its Nodetype value can be easily determined in constant time. For each internal node x with at least one absent child, the absent children are created and their values are determined as follows:

(a) An absent child R adjacent to a nonabsent child Q is selected. The color of R is determined according to Lemma 2. However, the color of R is determined directly from the Nodetype value of Q rather than from the leaf q referred to in Lemma 2. All other absent children of x are assigned the same color as R (Lemma 1).

(b) The Nodetype values of the previously absent children are determined. Finally, the Nodetype value of x is computed.

(2) From each leaf, a message is sent to the root of the tree. The messages move synchronously upward from level to level. (This is ensured by wait loops for messages starting at leaves of smaller depth.) At each level, the following is executed:

If all four messages reaching a node x come from black {white} children, then x is set to black {white} and its children are marked to be deleted. If the messages reaching x are from children with different color, then x is set to gray.

(3) All nodes marked to be deleted are removed, the remaining nodes are compressed to form a consecutive sequence, and all pointers are updated.

THEOREM 2. *The pointer-based quadtree representation of a binary image described by a boundary code of length b can be computed in time $O(h \log b)$ on a hypercube with $s = p = b$.*

Proof. The correctness of the algorithm follows from [16]. What remains to be shown is that the individual steps listed in the above two phases can be implemented with the claimed time complexity. We start with describing the hypercube implementation of *Phase* 1. For *Step* 1, the x coordinates of the absolute addresses are computed by assigning a value 1, -1, 0, and 0 to the boundary elements r, l, u, d, respectively, and computing the partial sums of this sequence. All y coordinates are computed analogously. For each boundary element, the creation of the boundary pixels requires only information about the directly adjacent boundary elements; otherwise, it is a local $O(1)$ time operation. Hence, Step 1 can be executed in $O(\log b)$ time. *Step* 2 requires $O(\log b)$ local computation steps at each processor. *Step* 3 requires time $O(\log b)$ [14]. *Step* 4 can be implemented by building the tree level by level, starting with the leaves (which are given). At each level, every node (initially leaves) examines its three neighbors to the right and left

and determines (using the shuffled row-major numbering the current level information) with whom a common ancestor is to be created. This can be implemented on the hypercube with $O(\log b)$ time per level, by using a constant number of partial sum as well as concentrate and distribute [14] operations. At the beginning of *Phase* 2, we have a quadtree template representing only the boundary pixels of the image I. The nodes corresponding to the black and white area inside and outside the boundary line, respectively, are now created by successive dynamic multiway search procedures. In *Step* 1, a dynamic multiway search procedure is used to add and update the absent children [cost: $O(h \log b)$]. Step 2 and Step 3 are respectively the same as Step 4 and Step 5 of the algorithm in Section 3.1. Therefore, Step 2 can be implemented in time $O(h \log b)$ and Step 3 requires time $O(\log b)$. ■

Linear quadtrees without path encoding can be constructed in essentially the same way by marking in Step 2 of Phase 1 also gray nodes as to be deleted. For linear quadtrees with path encoding, we also need to compute (between Steps 2 and 3 of Phase 2) the path encoding for each leaf by applying one additional multiway search procedure. Therefore, the linear quadtree representation (with or without path encoding) of a binary image represented by a boundary code of length b can also be computed in time $O(h \log b)$ on a hypercube with $s = p = b$.

4. HYPERCUBE ALGORITHMS FOR OPERATIONS ON QUADTREE

Numerous region properties of images such as the area or centroid, which are simply associative functions of the leaves (and do not need neighboring information), can be immediately calculated by partial sum operations (see [2]). This requires time $O(\log N)$ on a hypercube with $s = N$ and $p = N$.

4.1. *Finding Neighbors in Quadtrees and Computing Region Properties*

One of the main advantages of using the pointer-based quadtree is that, once the quadtree has been constructed, parallel searching algorithms on quadtrees can be easily adapted from the existing sequential methods by using the dynamic multiway search technique outlined in Section 2.3. One of the most important building blocks of quadtree applications are neighbor finding techniques. For a leaf x representing a quadrant X, a *neighbor* of x is a leaf y representing a quadrant that is adjacent to X (with respect to the image) and has at least the same size as X [17]. The *multiple neighbor-finding problem* consists of finding the neighbors of all leaves of the quadtree.

LEMMA 3. *Given a pointer-based quadtree of size N stored on a hypercube with $s = p = N$, then the multiple neighbor finding problem can be solved in time $O(h \log N)$.*

Proof. The sequential method described in [17] for finding the neighbor y of one single leaf x traverses the tree from x upward, along path $\pi(x)$, to the lowest common ancestor of x and y; then it descends downward to y by using the "mirror image" of the upward path $\pi(x)$. The main problem with parallelizing this method to parallel traversals for all leaves of the tree, using multiway search, is that a message used in multiway search may only be of constant size and, thus, cannot store the path $\pi(x)$. Assume w.l.o.g. that the right neighbor of x is to be determined. Let α denote the right border of the quadrant associated with x, and let β denote the line defined by extending α. We observe that a query can also be routed from a leaf x to its right neighbor y (along the same path as described above) as follows: The query moves upward from x until it reaches a node whose associated quadrant intersects β. Then it descends downward by selecting always the child whose associated quadrant is adjacent to α. Hence, a query process to be routed from x to its neighbor y needs to store only α and β. With this, multiple neighbor finding reduces to multiway search and, thus, the theorem follows. ■

Once the neighbors of each leaf in all four directions have been determined, the calculation of, e.g., the perimeter of the image follows immediately (see [2]).

THEOREM 3. *Given a pointer-based quadtree of size N stored on a hypercube with $s = p = N$, then the perimeter of the associated image can be computed in time $O(h \log N)$.*

4.2. *Rotating Quadtrees by 90°*

Given a pointer-based quadtree T, the following algorithm computes the quadtree T' for the image of T rotated by 90° on a hypercube or PRAM, with $s = p = N$.

(1) For each node, the position of the rotated associated quadrant is computed.

(2) For each rotated quadrant, the shuffled row-major number (with respect to the partitioning into quadrants of the same size) is computed.

(3) The nodes are sorted by major key *level* and minor key SRM number.

(4) All nodes are resorted to their original position in the old tree. Each node sends its new address to its parent.

(5) All nodes are again sorted by major key level and minor key SRM number.

THEOREM 4. *Given a pointer-based quadtree T of size N stored on a hypercube or PRAM with $s = p = N$, then the quadtree T' representing the image, associated with T, rotated by 90°, can be computed in time $O(h + \log N \log^2 \log N)$ and $O(h + \log N)$, respectively.*

Proof. The correctness of the algorithm follows from the observation that if a node v is the parent of a node w in T then the node in T' representing the rotated quadrant of v is also the parent of the node in T' representing the rotated quadrant of w. The computation of the shuffled row-major number in Step 2 requires $O(h)$ local computation steps at each processor. The remainder of the algorithm reduces to a constant number of sorting operations. Therefore, the time complexities follow. ∎

4.3. *Constructing Union and Intersection*

The union (intersection) of two quadtrees T_A and T_B is defined as the quadtree $T_{A\cup B}$ ($T_{A\cap B}$) representing the image composed of the bitwise OR (AND) of the two original images. In this section, we study the parallel computation of the union and the intersection of two pointer-based quadtrees. A work-optimal $O(\log N)$ time PRAM algorithm with $s = N$ and $p = N/\log N$ was introduced in [11]. The hypercube algorithm we will describe here represents an improvement on the hypercube simulation of this algorithm. Below, we introduce some definitions that will be used in the remainder of this section.

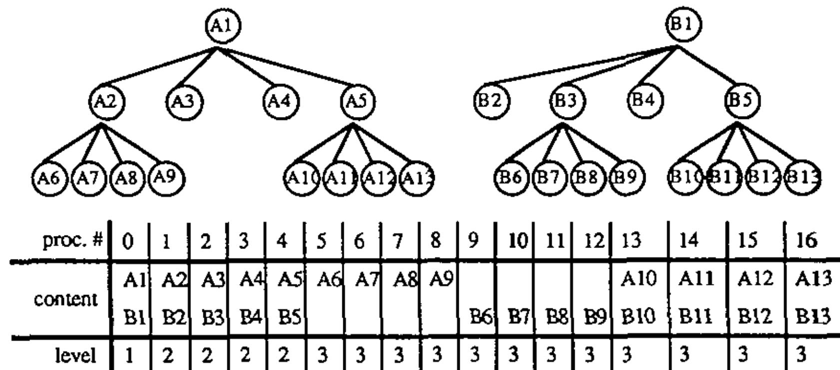
A tree T_{A+B} is called an *overlay* of T_A and T_B if it is the smallest 4-ary tree such that for each node v of T_A or T_B there exists a node $\delta(v)$ in T_{A+B} representing the same image area (assuming that T_{A+B} represents an image subdivision defined in standard quadtree fashion). The *combined level order numbering* of T_A and T_B is defined as follows: For each node v of T_A or T_B, the combined level order number $\eta_{A+B}(v)$ is the level order number of $\delta(v)$ in T_{A+B}. The *shuffled row-major number of a node v of T_A (or T_B)* is the shuffled row-major number of the associated quadrant with respect to the subdivision of the image plane into quadrants of the same size.

We assume that both quadtrees are stored by level order number as indicated in Section 2.2. As a preprocessing, we convert this storage scheme into a *combined level order numbering scheme*, where every node v of T_A or T_B is stored at processor number $\eta_{A+B}(v)$. Note that every processor stores at least one node, but at most two nodes, one of each tree. The new relative order of the nodes of one tree, say T_A, is the same as their order in the initial level order numbering of T_A. The combined level order numbering scheme can be obtained as follows: All nodes are permuted inside their levels according to their SRM number. For any two nodes within the same level that have the same SRM number and are stored in two adjacent processors P_i and P_{i+1}, the node in P_{i+1} is moved to P_i. Finally the contents of the processors are shifted leftward so that processors without data are avoided.

Given this storage scheme for the two quadtrees T_A and T_B, the following is an outline of a parallel algorithm for computing the quadtree $T_{A\cup B}$. Out algorithm uses dynamic multiway search (see Section 2.3) with three different types of messages: "compare," "copy," and "update" messages.

(1) From each of the roots of T_A and T_B a wave of compare messages is sent toward the leaves. That is, a compare message is sent to each root and, each node receiving a message duplicates it and sends one to each child (within its own tree). Messages move synchronously downward from level to level. During this process, a new tree T is created, which will subsequently be converted into $T_{A\cup B}$. At each level, the following is executed:

(a) Each node x receiving a compare message compares itself with the respective node y (representing the same image area) of the other tree. The node y is stored at the same processor P as node x and receives a compare



proc. #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
content	A1 B1	A2 B2	A3 B3	A4 B4	A5 B5	A6	A7	A8	A9	B6	B7	B8	B9	A10 B10	A11 B11	A12 B12	A13 B13
level	1	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3

FIG. 4. Combined level order numbering scheme.

message at the same time as node x does. Unless x and y are the roots of T_A and T_B, respectively, let *parent*(x) and *parent*(y) denote their respective parents. Note that, *parent*(x) and *parent*(y) are both gray nodes stored at the same processor P' and, previously, received a compare message at the same time.

Case 1. x and y are both gray. A new gray node z for T representing the same quadrant as x and y is created and stored at processor P. Note that *parent*(x) and *parent*(y) previously created a gray node z' for T. This node z' is made the parent of z in T.

Case 2. x or y is black. A new black node z for T representing the same quadrant as x and y is created and stored at processor P. The gray node z' created by *parent*(x) and *parent*(y) is made the parent of z in T. The two compare messages which reached x and y are not forwarded but deleted.

Case 3. One node, x or y, is gray and the other node is white. A new gray node z for T representing the same quadrant as x and y is created and stored at processor P. The gray node z' created by *parent*(x) and *parent*(y) is made the parent of z in T. The compare message which reached the white node is deleted. The compare message which reached the gray node is changed to a copy message, duplicated, and forwarded to all children.

Case 4. x and y are both white. A new white node z for T representing the same quadrant as x and y is created and stored at processor P. The gray node z' created by *parent*(x) and *parent*(y) is made the parent of z in T. The two compare messages which reached x and y are not forwarded but deleted.

(b) Each node x receiving a copy message (in the other tree there exists no node y representing the same quadrant) creates a new node z for T with the same color as x and representing the same quadrant. The node z' created by *parent*(x) and *parent*(y) is made the parent of z in T. A copy message is sent to each child, or the message is deleted if x is a leaf.

(2) From each leaf an update message is sent to the root of the tree. The update messages move synchronously upward from level to level. (This is ensured by wait loops for messages starting at leaves of smaller depth.) At each level, the following is executed:

If all four update messages reaching a node x come from black {white} children, then x is set to black {white} and its children are marked to be deleted. If the update messages reaching x are from children with different color, x is set to gray.

(3) All nodes marked to be deleted are deleted, the remaining nodes are compressed to form a consecutive sequence, and all pointers are updated.

Computing the intersection of two pointer-based quadtrees is analogous. All steps of the above algorithm remain unchanged except for Cases 2–4 where black and white should be exchanged.

THEOREM 5. *Given two pointer-based quadtrees with a total number of N nodes stored on a hypercube with $s = p = N$, then the union {intersection} of these quadtrees can be computed in time $O(h \log N)$, where h denotes the maximum height of the two trees.*

Proof. In order to observe the correctness of the algorithm we first study the intermediate tree T created at the end of Step 1. Consider two nodes x and y in T_A and T_B representing the same quadrant. Then a node z in T is created in Step 1a (a compare message reaches x and y), and it is easy to see that through Cases 1 to 4 the right color, representing the union {intersection} of x and y, is assigned to z. Consider, on the other hand, a node x for quadrant X in, say, T_A with no node in T_B representing the same quadrant. Then T_B has a leaf y for a quadrant Y containing X. Let x' be the ancestor of x representing quadrant Y. If Y is black {white} then no node needs to be created in T, which is guaranteed by the deletion of the compare messages reaching x' and y (Step 1a, Case 2). If Y is white {black} then the entire subtree rooted at x' has to be copied into T. This is achieved by the copy messages started at x' (Step 1a, Case 3 and Step 1b).

In order to prove the claimed time complexity, we first observe that the preprocessing reduces to a constant number of partial sum, concentrate, and distribute operations [14]. Hence, its time complexity is $O(\log N)$. The combined level order numbering scheme used to store the trees T_A, T_B, and T allows simultaneous multiway search on all three trees, because T_A, T_B, and T are subtrees of T_{A+B}, and all nodes are stored with respect to their level order number in T_{A+B} (see Section 2.2 and 2.3). Hence, Step 1 can be implemented on a hypercube using the dynamic multiway search procedure outlined in Section 2.3. That is, Step 1 requires time $O(h \log N)$. Steps 2 and 3 are equivalent to Steps 4 and 5 of the algorithm in Section 3.1. Hence, from Theorem 1, their time complexity is $O(h \log N)$. ∎

5. SOME OBSERVATIONS FOR THE PRAM

A (CREW) PRAM consists of a set $P_1, \ldots, P_p$ of p processors, with constant size local memories, connected to a shared memory of size s. An arbitrary number of processors can read concurrently from the same shared memory location, but concurrent write accesses are not possible.

Some of our results can be immediately extended to the PRAM. A straightforward implementation of multisearch (Section 2.3) on the PRAM, together with Theo-

rems 2–4, yields the following observations for the PRAM.

Observation 1. The pointer-based quadtree representation of a binary image described by a boundary code of length b can be computed in time $O(h \log b)$ on a PRAM with $s = p = b$.

Observation 2. Given a pointer-based quadtree of size N stored on a PRAM with $s = p = N$, then the perimeter of the associated image can be computed in time $O(h)$.

Observation 3. Given a pointer-based quadtree T of size N stored on a PRAM with $s = p = N$, then the quadtree T' representing the image, associated with T, rotated by 90° can be computed in time $O(h)$.

Note that the emphasis of this paper is on hypercube algorithms, and these straightforward observations for the PRAM are not necessarily optimal. They are listed only because either no previous PRAM algorithm existed (Observations 1 and 3) or they give an improvement on existing results (Observation 2).

6. CONCLUSION

In this paper we have demonstrated that, for parallel processing, pointer-based quadtrees are an attractive alternative to linear quadtrees. We presented efficient hypercube algorithms for constructing and manipulating quadtrees. These algorithms can be easily implemented in hypercube-based SIMD parallel machines.

REFERENCES

1. J. L. Bentley and D. F. Stanat, Analysis of range searches in quad trees, *Inform. Process. Lett.* **3**(6), 1975, 170–173.
2. S. K. Bhaskar, A. Rosenfeld, and A. Y. Wu, Parallel processing of regions represented by linear quadtrees, *Comput. Vision Graphics Image Process.* **42**, 1988, 371–380.
3. R. Cole Parallel merge sorting, *SIAM J. Comput.* **17**(4), 1988, 770–785.
4. R. Cypher and C. G. Plaxton, Deterministic sorting in nearly logarithmic time on a hypercube and related computers, in *Proceedings 22nd Annual ACM Symposium on Theory of Computing*, 1990, pp. 193–203.
5. F. Dehne, A. Ferreira, and A. Rau-Chaplin, Parallel branch and bound on fine grained hypercube multiprocessors, *Parallel Comput.*, to appear.
6. F. Dehne and A. Rau-Chaplin, Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry, *J. Parallel Distrib. Comput.* **8**, 1990, 367–375.
7. S. Edelman and E. Shapiro, Quadtrees in concurrent prolog, in *Proceedings International Conference on Parallel Processing, 1985*, pp. 544–551.
8. R. A. Finkel and J. L. Bentley, Quad trees—A data for retrieval on composite keys, *Acta Inform.* **4**(1), 1974, 1–9.
9. Y. Hung and A. Rosenfeld, Parallel processing of linear quadtrees on a mesh-connected computer, *J. Parallel Distrib. Comput.* **7**, 1989, 1–27.
10. K. J. Jacquemain, The complexity of constructing quad-trees in arbitrary dimensions, in *Proceedings 7th Conference on Graphtheoretic Concepts in Computer Science (WG81), 1982* (J. Mühlbacher, Ed.), pp. 293–301.
11. S. Kasif, Efficient parallel quad-tree algorithms, *Proceedings of the 1988 ICAI, Tel Aviv, Israel, 1988*, pp. 353–363.
12. M. Martin, D. M. Chiarulli, and S. S. Iyengar, Parallel processing of quadtrees on a horizontally reconfigurable architecture computing system, in *Proceedings International Conference on Parallel Processing, 1986*, pp. 895–902.
13. G.-G. Mei and W. Liu, Parallel processing for quadtree problems, in *Proceedings International Conference on Parallel Processing, 1986*, pp. 452–454.
14. D. Nassimi and S. Sahni, Data broadcasting in SIMD computers, *IEEE Trans. Comput.* **30**(2), 1981, 101–106.
15. D. Nassimi and S. Sahni, A self-routing Benes network and parallel permutation algorithms, *IEEE Trans. Comput.* **30**(5), 1982, 332–340.
16. H. Samet, Region representation: Quadtrees from boundary codes, *Commun. ACM* **23**(3), 1980, 163–170.
17. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Comput. Graphics Image Process.* **18**(1), 1982, 37–57.
18. H. Samet, The quadtree and related hierarchical data structures, *Comput. Surv.* **16**(2), 1984, 187–260.