

Syracuse University

SURFACE

Electrical Engineering and Computer Science -
Technical Reports

College of Engineering and Computer Science

11-1990

Hypercube Algorithms for Operations on Quadrees

Ravi V. Shankar
Syracuse University

Sanjay Ranka
Syracuse University

Follow this and additional works at: https://surface.syr.edu/eecs_techreports



Part of the [Computer Sciences Commons](#)

Recommended Citation

Shankar, Ravi V. and Ranka, Sanjay, "Hypercube Algorithms for Operations on Quadrees" (1990). *Electrical Engineering and Computer Science - Technical Reports*. 83.

https://surface.syr.edu/eecs_techreports/83

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science - Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

SU-CIS-90-34

***Hypercube Algorithms for Operations
on Quadrees***

Ravi V. Shankar and Sanjay Ranka

November 1990

*School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100*

SU-CIS-90-34

Hypercube Algorithms for Operations on Quadrees

Ravi V. Shankar and Sanjay Ranka

November 1990

*School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, NY 13244-4100*

(315) 443-2368

Hypercube Algorithms for Operations on Quadrees

Ravi V Shankar and Sanjay Ranka *
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-4100

2 November 1990

Abstract: This paper describes parallel algorithms for the following operations on quadrees - boolean operations (union, intersection, complement), collapsing a quadtree, and neighbor finding in an image represented by a quadtree. The architecture assumed in this paper is a hypercube with one processing element (PE) per hypercube node. We assume that the architecture is SIMD, i.e., all PEs work under the control of a single control unit.

Keywords: Quadrees, Hypercube algorithms, Image processing

1 Introduction

A quadtree[1] is a tree representation of a sparse image (in general, any 2D array). The root of the quadtree represents the entire image. If the portion of the image represented by any node does not have the same gray value, the node is assigned four children. Each child represents one of the four quadrants of the image portion represented by its parent. This continues recursively until all the leaf nodes represent portions of the image with the same gray value.

Throughout this work we assume that the input image is binary. The algorithms can be extended to deal with gray-level/color images.

1.1 Definitions

The *level* of a quadtree node is its distance (in terms of number of links) from the root. The *height* of a quadtree is the greatest of the distances between the nodes of the quadtree and the root. The term node actually indicates a collection of pixels. A node at level l in a quadtree of height h represents a collection of $4^{(h-l)}$ pixels.

Every non-leaf node in the quadtree has exactly four children. Every leaf node is either filled (i.e., has a value of 1) or empty (i.e., has a value of 0).

Nodes representing single pixels have the same index as the shuffled row-major index of the pixel they represent. The index of nodes representing a collection of pixels is the smallest of the indices of the pixels represented. Shuffled row-major indexing for an 8x8 image is shown in figure 1a.

A node X in tree T1 is said to *cover* a node Y in tree T2 if (i) there exists a node Z in T2 such that Z is an ancestor of Y and $\text{index}(Z)$ and $\text{level}(Z)$ are equal to $\text{index}(X)$ and $\text{level}(X)$ respectively, or (ii) X is identical to Y.

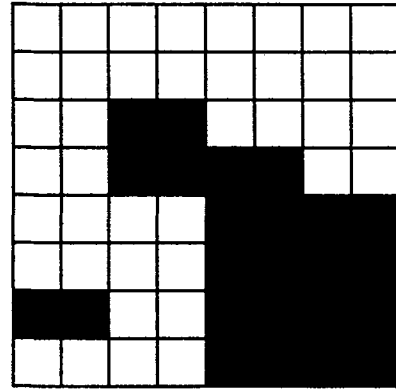
1.2 Representation of the quadtree

There are many ways in which a quadtree can be stored. The most expensive method is to store the actual tree including the values and the pointers at each node. A better way would be to store only the leaves of the quadtree along with the corresponding indices and values. Even this contains redundant information since information about empty leaves can be obtained given the information about filled leaves alone. We represent the quadtree using its only its filled leaves. Given a quadtree with N filled nodes, either $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$ are stored in each PE (P = number of PEs). The index, and level of each filled leaf are stored in the *index* and *level* registers. A additional *value* register is needed for gray level/color images.

Figures 1(b) and 1D(c) shows a binary image, a quadtree for the same image, and the 1D representation of the quadtree. The 1D representation of quadtrees is referred to as “linear quadtrees” in the literature.

The quadtree representation in a 1D array of PEs is said to be in standard form when (i) only filled leaf nodes are represented (ii) the filled leaf nodes are arranged with their indices in increasing order, and (iii) every PE to the left of a PE having a filled leaf node has a filled leaf node of its own. All algorithms in this paper assume that the input is in standard form. The

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



Index: 12 26 27 40 41 48
Level: 2 3 3 3 3 1

Figure 1a Shuffled Row-Major Indexing for an 8x8 Image

Figure 1b A Binary Image and Its 1D Representaion

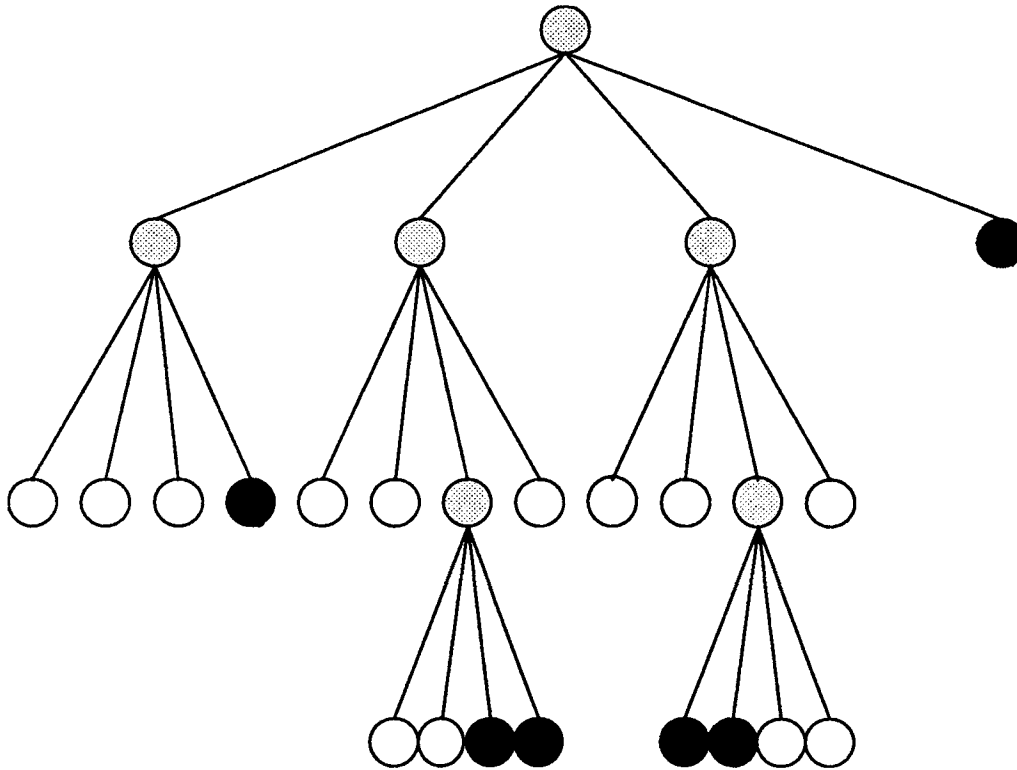


Figure 1c A Quadtree for figure 1a

output of all algorithms is also in standard form.

1.3 Earlier work

Sequential algorithms for processing pointer-less quadtrees are described in [2] and [3]. Parallel quadtree algorithms for various architectures can be found in the literature. Mei and Liu[4] consider quadtree algorithms for 2D shuffle exchange network. The paper also assumes a static allocation of all image pixels (empty and filled) to processors, which in the case of sparse images would result in a lot of idle processors. Martin et al [5] describe algorithms for a horizontally reconfigurable architecture. Hung and Rosenfeld[6] consider a mesh connected computer. It appears that their intersection/union algorithm assumes the availability of both empty as well as filled leaves, which represents redundant input. Our collapse algorithm is based on one of their quadtree building algorithms, while our neighbor finding algorithm is a modified version of their algorithm for the hypercube. Nandy et al [7] describe linear quadtree algorithms for neighbor finding and boundary following on an MIMD hypercube. Their work does not give complexity analysis for the complete embedding of the quadtree.

2 Hypercube Primitives

2.1 Concentrate

In the Concentrate algorithm we start with a subset of the processing elements, each containing data in register D, and the PE's rank (that is, the number of selected PEs with lower index than self) in register R. The objective is to move the data in register D such that $D(i)$ goes to the PE with index $R(i)$. This primitive is used to bring the 1D representation of a quadtree into standard form.

The Concentrate algorithm is described in [8]. Figure 2(a) illustrates the concentrate operation. The time complexity of the algorithm is $O((N/P) \log P)$ where N is the size of the given input and P is the no of PEs.

2.2 Merge

Merging of two sorted arrays can be done on the hypercube using the bitonic merge algorithm. The hypercube algorithm is described in [9]. The merge algorithm takes time $O((N/P)\log P)$. The merge primitive is used in the quadtree intersection/union algorithm.

2.3 Generalize

In the Generalize algorithm we start with data in register D in the first k PEs. A destination PE index is available in register R and is such that $R(i-1) < R(i)$ for $0 \leq i \leq k$. For convenience, assume $R(-1) = 0$. The objective is to move the data in register D such that $D(i)$ goes to all the PEs with index k satisfying $< R(i)$ and $\geq R(i-1)$. This primitive is used to obtain the complement of a quadtree.

The Generalize algorithm is described in [8]. Figure 2(b) illustrates the generalize primitive. The time complexity of the algorithm is $O((N/P)\log P)$.

2.4 Segmented Scans

In the segmented prefix scan algorithm a 1-bit register S is used to indicate the start of a new segment when set to 1. Data is available in register D . A binary associative operator \oplus is specified. The objective is to obtain in PE i the quantity $D(j) \oplus D(j+1) \oplus \dots \oplus D(i)$ where j satisfies the following properties (i) $j \leq i$ (ii) $S(j) = 1$ and (iii) for all k satisfying $j < k \leq i$ and $S(k)=0$. Segmented scans are used in the quadtree union/intersection, collapse and in the neighbor finding algorithms.

The segmented scan algorithm is a modified form of the prefix scan algorithm presented in [8]. Figure 2(c) illustrates the scan primitive. The time complexity of the algorithm is $O((N/P) + \log P)$.

2.5 Sort

Bitonic sort can be used to sort an array on the hypercube. The sorting algorithm is described in [9]. The bitonic sort algorithm takes time $O(\log^2 N)$ when $P=N$. A faster deterministic sorting algorithm that runs in nearly logarithmic time is presented in [10]. The complexity of this sorting algorithm is

D	-	d ₁	-	-	d ₄	d ₅	d ₆	-
R	α	0	α	α	1	2	3	α
D(after Concentrate)	d ₁	d ₄	d ₅	d ₆	-	-	-	-

(a) Concentrate

D	d ₀	d ₁	d ₂	d ₃	-	-	-	-
R	2	3	6	7	α	α	α	α
D(after Generalize)	d ₀	d ₀	d ₀	d ₁	d ₂	d ₂	d ₂	d ₃

(b) Generalize

D	7	9	4	6	8	2	1	5
D(after +-scan)	7	16	20	26	34	36	37	42
S	1	0	0	1	0	0	1	0
D(after segmented +-scan)	7	16	20	6	14	16	1	6

(c) Non-segmented / Segmented Prefix Scan

Figure 2. Hypercube primitives

$O(\log N(\log \log N)^2)$ when $P=N$. $Sort(N, P)$ is used throughout this paper to indicate the time taken to sort N elements on a hypercube with P PEs. The Sort primitive is used in the neighbor finding algorithm.

3 Boolean Operations

3.1 Intersection/Union

The intersection/union of two quadtrees T_1 and T_2 is a quadtree T such that the image represented by T is the intersection/union of the images represented by T_1 and T_2 .

The intersection/union algorithm is outlined in figure 3.

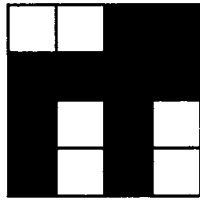
Figure 4 illustrates the intersection/union algorithm through an example. The *Index*, *Level*, and *Tree-no* registers are set by merging the given quadtrees as mentioned in step 1 of the algorithm. Initially the *Cov* register is set as described in step 2 - a '1' indicates a covering node, '0' a covered node, '9' stands for a node to be deleted, '8' for a node that will definitely appear in the resulting tree, and '7' for a node that hasn't been labeled yet. The contents of the *Cov* register are modified after determination of 'leaders'. The label '4' indicates that a node is covered by its leader and '5' indicates that it is not. The resulting tree after the intersection and union operations are obtained as described in step 5. These are available in registers *Inter* and *Union*.

For the intersection/union algorithm to work correctly we only need to correctly mark all the filled leaves in the given trees as 'covering', 'covered by a filled leaf' or 'covered by an empty node'. This is because we can retain all nodes covered by an empty leaf and remove those covered by a filled leaf for quadtree union. For intersection, we can retain all nodes covered by a filled leaf while removing the filled cover of those nodes and also remove nodes covered by an empty leaf.

The correctness of the union/intersection algorithm can be proved as follows. In step 2, cases a,b, and c mark the covering nodes in the input quadtree correctly. Steps 3,4, and 5 mark the quadtree nodes that are covered by a filled leaf. We claim that the nodes in the given quadtrees that haven't been marked so far are all covered by an empty leaf. This is true since all nodes covered by any leaf X appear as a continuous run of nodes after X in

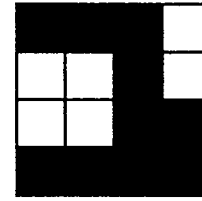
-
1. Merge trees T_1 and T_2 such that the merged tree is sorted by $\langle \text{index}, \text{level}, \text{tree-no} \rangle$
 2. Let each node P examine its immediate successor X . The following cases may arise:
 - a $\text{index}(P) = \text{index}(X)$, $\text{level}(P) = \text{level}(X)$, $\text{tree-no}(P) < \text{tree-no}(X)$
 - b $\text{index}(P) = \text{index}(X)$, $\text{level}(P) < \text{level}(X)$
 - c $\text{index}(P) < \text{index}(X)$
- Case a P covers X and X covers P . Mark one of the nodes (say P) as 'covering' and mark the other for deletion.
- Case b $\text{tree-no}(P)$ must be different from $\text{tree-no}(X)$ since no two filled leaves from the same tree will have identical indices. P covers X . Mark P as 'covering' and X as 'covered'.
- Case c If $\text{index}(X) < \text{index}(P) + \text{size}(P)$, then P covers X . Mark P as 'covering' as X as 'covered'.
3. Split the nodes into segments with the covering nodes determined so far marking the start of new segments.
 4. The index and level of the covering nodes (we will call these 'leaders') are copied onto each node in the segment.
 5. Each node in the segment checks whether it is covered by its leader. If yes, leave as such for intersection and invalidate the leader. For union remove these. If not, it is covered by an empty node. Remove these for intersection. Leave as such for union.
 6. *Concentrate* to remove all nodes that were invalidated or marked for deletion.

Figure 3 Algorithm for the intersection/union of two quadtrees



Index: 2 3 4 8 10 12 14
Level: 2 2 1 2 2 2 2

(a) Tree T_1

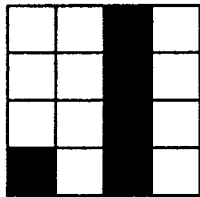


Index 0 1 4 6 10 11 12
Level 2 2 2 2 2 2 1

(b) Tree T_2

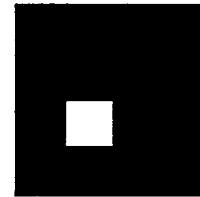
<i>Index:</i>	0	1	2	3	4	4	6	8	10	10	11	12	12	14
<i>Level:</i>	2	2	2	2	1	2	2	2	2	2	2	1	2	2
<i>Tree-no:</i>	1	1	0	0	0	1	1	0	0	1	1	1	0	0
<i>Coverl:</i>	7	7	7	7	1	0	0	7	7	9	7	1	0	7
<i>Cov:</i>	5	5	5	5	1	4	4	5	9	8	5	1	4	4
<i>Inter:</i>	0	0	0	0	0	1	1	0	0	1	0	0	1	1
<i>Union:</i>	1	1	1	1	1	0	0	1	0	1	1	1	0	0

(c) Steps in the Union/Intersection Algorithm.



Index: 4 6 10 12 14
Level: 2 2 2 2 2

(d) Tree $T_1 \cap T_2$



Index: 0 4 8 10 11 12
Level: 1 1 2 2 2 1

(e) Tree $T_1 \cup T_2$

Figure 4. Union/Intersection

-
1. Each node looks at its immediate successor and finds out the number of empty pixels between between them.
 2. The *generalize* algorithm is used to spread the empty nodes one per PE.
 3. A *collapse* algorithm is run to group empty nodes together and to bring the result to standard form.
-

Figure 5 The Complementing algorithm

the sorted ordering described in step 1.

This is because empty covering nodes are not available in our representation. Further, all nodes covered by P weren't marked as 'covered' in cases b and c.

A node X is covered by its leader Y iff $\text{index}(Y) = \text{index}(X) - \text{index}(X) \bmod 4^{\text{height} - \text{level}(Y)}$. In step 5, invalidation of the leader can be done by setting up an invalidation request and finally performing a segmented backward OR scan to do the actual invalidation.

The merge in step 1 and the concentrate in step 6 take time $O((N/P) \log P)$. The prefix scan in steps 4 and 5 take $O((N/P) + \log P)$ time. Steps 2 and 3 take $O(N/P)$ time. Thus the intersection/union algorithm has an $O((N/P) \log P)$ time complexity.

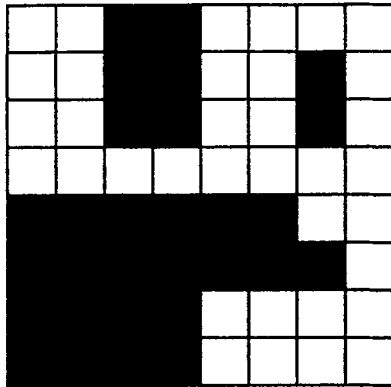
3.2 Complement

The complement algorithm is outlined in figure 5.

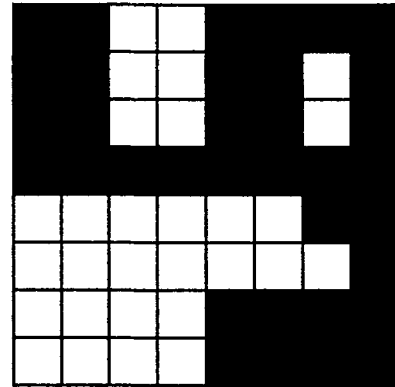
Let N_1 be the number of filled leaves in the given quadtree. Step 1 takes $O(N_1/P)$ time and steps 2 and 3 take $O((N/P) \log P)$ time, where N is the sum of sizes (in pixels) of the empty leaves in the given quadtree. The entire complement algorithm has an $O((N/P) \log P)$ time complexity. The algorithm, however, has a high worst-case space complexity.

The number of empty nodes is determined as follows: Let A and B be the 2 leaf nodes between which we wish to find the number of empty nodes. Find the index i_1 of the rightmost node at the lowest level of the subtree rooted at A and the index i_2 of the leftmost node at the lowest level of the subtree rooted at B. $(i_2 - i_1 - 1)$ gives the number of empty nodes between A and B. (Minor modifications can be made for the first and the last leaf nodes).

To reduce the amount of space required the following modification can



(a) Tree T_3



(b) Complement of T_3

Index: 4 12 13 22 28 32 48 54
Level: 2 3 3 3 3 1 2 3

(c) 1D Representation of T_3

Index: 4 12 13 22 28 32 48 54
Level: 3 3 3 3 3 1 2 3
I1: 4 12 13 22 28 32 48 54 (64)
I2: (-) 7 12 13 22 28 47 51 54
NumEmpty: 4 4 0 8 5 3 0 2 9
IndOfEmpty: 0 1 2 3 8 9 10 11 14 15 16 17 18 20 21 23
 24 25 26 27 29 30 31 52 55 56 57 58 60 61 62 63

(d) Steps in the Complement Algorithm

Index 0 8 14 15 16 20 21 23 24 29 30 31 52 53 55 56 60
Level 2 3 3 3 2 3 3 3 2 3 3 3 3 3 3 2 2

(e) 1D Representation of complement of T_3

Figure 6 Complement Algorithm.

be used. Let l_1 and l_2 be the levels of nodes A and B. If $l_2 > l_1$ find the index of the ancestor of B in level l_1 (Call this ancestor C). A notes down the level number l_1 along with the number of nodes between A and C, and level l_2 along with the number of nodes to the left of B having C as an ancestor. If $l_1 \geq l_2$ let D be the ancestor of A at level l_2 . Now B notes down level l_2 along with the number of nodes between D and B and level l_1 along with number of nodes to the right of A having D as an ancestor. A *generalize* and a *compact* can now be done as in the earlier case. The time complexity is still $O((N/P) \log P)$ although the space required has been reduced.

The first method is illustrated in Figure 6.

3.3 Difference

The difference operation between two trees can be carried out by complementing the second tree, followed by an intersection between the resulting tree and the first tree. The time taken by difference is $O((N/P) \log P)$.

4 Collapsing the tree

A quadtree needs to be collapsed when all the four children of a non-leaf node have identical values. Such nodes are redundant and can be removed after the contents of their *value* register is passed on to their parent node.

Figure 7 gives the steps in the collapse algorithm.

Steps 1,2,4,5 and 6 of the collapse algorithm take $O(N/P)$ time. Step 3 takes time $O((N/P) + \log P)$ and step 7 takes $O((N/P) \log P)$ time. Thus, the collapse algorithm has a worst case time complexity of $O((N/P) \log P)$.

Figure 8 illustrates the collapse algorithm.

5 Neighbor Finding

Our standard form for the representation of the quadtree stores the blocks in the image in Shuffled Row Major (SRM) order. We begin with this representation and compute the North, South, East, and West neighbors of each block. Note that any node node looks only for nodes smaller than itself. The steps in the East neighbor finding algorithm are outlined in figure 9.

-
1. Each node determines the maximum number of pixels (filled or empty) it can represent. This is stored in register *NumPix*.
 $NumPix(i) = 4^{NTZP(i)}$ where $NTZP(i)$ is the number of trailing zero-pairs in PE i .
 2. The filled leaves of the quadtree are split into segments. A 1-bit register *segment* in each PE is used to indicate whether the node in that PE is the beginning of a segment. Register *segment* is set to true if the index of the node in that PE minus the index of the last pixel of the node in the PE immediately preceding it is greater than 1. The index of the last pixel in any node = index of node + size of node - 1 where size of a node = $4^{(height-level)}$
 3. Each node obtains the number of pixels preceding it in the same segment and stores it in register *position*. This is just a segmented sum scan of the size of the node each PE represents. 1 + the position of the last PE in a segment gives the segment's length. A segmented backward copy scan makes this value available to each node in the segment in register *length*.
 4. Each node stores in register *follow* the number of pixels following it (including self) in the same segment. This is just $length - position$. *Follow1* is set to the largest power of 4 \leq the contents of the *follow* register in the same PE.
 5. The minimum of *NumPix* and *follow* in each PE gives the size of the largest block of filled leaves represented by the node in that PE. This is available in the register *MaxBlkFL*. The register *level1* is set based on the contents of *MaxBlkFL*.
 6. Nodes that are redundant need to be deleted. A node is redundant when it represents only a subset of the pixels represented by another node. Using the index and level information each node first finds the number of its sibling nodes that have higher indices than self and the number of siblings with lower indices. By comparing this with the contents of the *position* and *follow* registers the node can determine whether it is redundant or not. If it is redundant it is marked for deletion.
 7. *Concentrate* to remove nodes that were marked for deletion.
-

Figure 7 Quadtree Collapse algorithm

<i>Index:</i>	0	1	2	3	4	8	10	11	12
<i>Level:</i>	1	2	2	2	1	2	2	2	1
<i>Size:</i>	1	1	1	1	4	1	1	1	4
<i>Seg:</i>	1	0	0	0	0	0	1	0	0
<i>Position:</i>	0	1	2	3	4	8	0	1	2
<i>Follow:</i>	9	8	7	6	5	1	6	5	4
<i>MaxBkSz:</i>	4	1	1	1	4	1	1	1	4
<i>Follow1:</i>	4	4	4	4	4	1	4	4	4
<i>NumPix:</i>	16	1	1	1	4	4	1	1	4
<i>Levell:</i>	1	2	2	2	1	2	2	2	1
<i>LSib:</i>	0	1	2	3	4	8	2	3	12
<i>Self+RSib:</i>	64	3	2	1	12	8	2	1	4
<i>Redun:</i>	0	1	1	1	0	0	0	0	0

(a) Steps in Collapsing tree $T_1 \cup T_2$ from figure 4.

<i>Index:</i>	0	4	8	10	11	12
<i>Level</i>	1	1	2	2	2	1

(b) 1D Representation of Collapsed Tree T_{new}

Figure 8 Collapsing a Tree

-
- 1) Each node computes its Column Major/Row major address from its index by bit shuffling. These are available in registers *CMAdd* and *RMAdd*.
 - 2) The nodes are then rearranged in increasing CM address order (Register *CMAdd1*). Note that all the East neighbors of any node will now appear as a continuous run. Each node just needs the Column Major addresses of its topmost and bottommost East neighbors. These addresses can be used to select the segment of nodes representing its East neighbors.
 - 3) The row major address of the topmost neighbor of any node X is the address of the immediate successor of X in Row Major order (This successor could be a filled or an empty node). This is available in register *RMSucc*.

$$RMSucc(i) = RMAdd(i) + \sqrt{size(X)}$$

- 4) The row major addresses in *RMSucc* are converted into column major addresses and stored in *Begin*.
 - 5) The highest possible column major address an East neighbor could have is computed and stored in *End*.
 - 6) The computed column major addresses from steps 4 and 5 are sorted, tagged, and merged with the sorted sequence from step 2. Segmented scans can now be used to collect information from the *data* registers of the East neighbors.
-

Figure 9 Neighbor finding algorithm

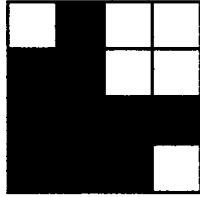
Modifications for North, South, and West neighbor finding should be easy. The algorithm can also be extended for 8-neighbor finding.

Time complexity of steps 1,3,4 and 5 of the Neighbor finding algorithm is $O(N/P)$. Steps 2 and 6 involve sorting and hence the neighbor finding algorithm takes $O((N/P) + Sort(N, P))$ time.

The neighbor finding algorithm is illustrated in figure 10.

6 Acknowledgement

The authors would like to thank Ravi Ponnusamy for his help with this work and for his comments on earlier versions of this paper.



<i>Index:</i>	1	2	3	4	8	12	13	14
<i>Level:</i>	2	2	2	2	1	2	2	2
<i>Data:</i>	1	2	3	4	5	6	7	8
<i>CMAdd:</i>	4	1	5	8	2	10	14	11
<i>CMAdd1:</i>	1	2	4	5	8	10	11	14
<i>Data1:</i>	2	1	5	3	4	6	8	7
<i>RMSucc</i>	2	5	6	3	10	11	16	15
<i>Begin:</i>	8	5	9	12	10	14	16	15
<i>End:</i>	8	5	9	12	11	14	16	15
<i>Neighb:</i>	4	3	0	0	14	7	0	0

Figure 10 Steps in the Neighbor Finding Algorithm.

References

- [1] Hunter, G.M., and Steiglitz, K., Operations on images using quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1, pp.145-153, 1979.
- [2] Gargantini, I., An effective way to represent quadtrees, *Communications of the ACM*, 25, pp.905-910.
- [3] Gargantini, I., Translation, rotation, and superimposition of quadtrees, *International Journal of Man-Machine Studies*, 18, pp.253-263, 1983
- [4] Mei, G.G., and Liu, W., Parallel processing for quadtree problems, *Proceedings International Conference on Parallel Processing*, pp.452-454, 1986.
- [5] Martin, M., Chiarulli, D.M., and Iyengar, S.S., Parallel processing of quadtrees on a horizontally reconfigurable architecture computing system, *Proceedings International Conference on Parallel Processing*, pp.895-902, 1986.
- [6] Hung, Y., and Rosenfeld, A., Parallel processing of linear quadtrees on a mesh-connected computer, *Journal of Parallel and Distributed Computing*, 7, pp.1-27, 1989.
- [7] Nandy, S.K., Moona, R., and Rajagopalan, S., Linear Quadtree algorithms on the hypercube, *Proceedings International Conference on Parallel Processing*, pp.227-229, 1988.
- [8] Nassimi, D., and Sahni, S., Data Broadcasting in SIMD Computers, *IEEE Transactions on Computers*, 30(2), pp.101-107, 1981.
- [9] Ranka, S., and Sahni, S., *Hypercube algorithms for Image Processing and Pattern Recognition*, Springer-Verlag, 1990.
- [10] Plaxton, G., and Cypher, R., 'Deterministic Sorting in nearly logarithmic time', *Proc. ACM Symposium on Theory of Computing*, 1990, pp.193-203.

- [11] Bestul, T., A general technique for creating SIMD algorithms on parallel pointer-based quadtrees, Technical Report CS-TR-2181, University of Maryland, College Park, 1989.