CVGIP: IMAGE UNDERSTANDING
Vol. 59, No. 3, May, pp. 281–285, 1994

Optimal Parallel Algorithms for Quadtree Problems

SIMON KASIF

Computer Science Department, The Johns Hopkins University, Baltimore, Maryland 21218

Received November 9, 1990; accepted September 24, 1993

In this paper we describe optimal processor-time parallel algorithms for set operations such as union, intersection, comparison on quadtrees. The algorithms presented in this paper run in $O(\log N)$ time using $N/\log N$ processors on a shared memory model of computation that allows concurrent reads or writes. Consequently they allow us to achieve optimal speedups using any number of processors up to $N/\log N$. The approach can also be used to derive optimal processor-time parallel algorithms for weaker models of parallel computation. © 1994 Academic Press, Inc.

1. INTRODUCTION

In this paper we address the problem of designing efficient parallel algorithms for computing set operations such as union, intersection, comparison, and matching on digital images represented by region quadtrees. These operations have wide applicability in computer vision, e.g., for image matching and image morphology. A region quadtree is a data structure for storing digital binary images. The structure is generated by a recursive decomposition of a nonhomogeneous image quadrant (block) of size 2^m by 2^m into four quadrants each of size 2^{m-1} by 2^{m-1}. The decomposition continues until all quadrants are either all black or all white homogeneous regions. The quadtree has numerous applications in computer vision, image processing, and computer graphics, where they use octrees. There is a substantial amount of literature on efficient processing of quadtrees on sequential machines (see [S84] for a comprehensive survey of the field). Several parallel algorithms for quadtree problems have been developed for various models of computation such as PRAMS, meshes, and hypercubes [BRW88, ES85]). Previous parallel algorithms for the problems considered in this paper (see references in [BRW88]) compute in time linear in the depth of the tree and thus, in the worst case do not improve on the sequential time complexity. In this paper we describe optimal logarithmic-time parallel algorithms for the standard tree representation of the quadtree. Several previous papers have considered linear quadtrees. Our approach yields $O(\log N)$ time complexity with $N/\log N$ processors (where N is the size of the quadtree). Therefore, the algorithms exhibit optimal speedup for any number of processors P, $P \leq N/\log N$ by the standard use of Brent's theorem [Bre74].

The algorithms described can be implemented on a CRCW PRAM (concurrent read concurrent write parallel RAM). This is a shared memory model that allows simultaneous reads and writes to the same memory location. When two or more processors attempt to write to the same location, only one of the write requests succeeds. This model of computation can be simulated in logarithmic time on weaker models such as EREW PRAM (exclusive read, exclusive write model) which prohibits concurrent reads or writes. Our algorithms can also be implemented with optimal speedup on the EREW PRAM (see discussion in Section 5).

2. PRELIMINARIES

For completeness, we give a brief introduction to region quadtrees in this section (see [S84] for a complete description of region quadtrees). A quadtree is an ordered tree, such that each internal node has four children. Leaf nodes are labelled with a binary label from the set $\{black, white\}$ and each internal node is assumed to have the label *grey*. We also associate a label with each edge. Edge labels are chosen from the set $\{0, 1, 2, 3\}$. In standard quadtree literature edges are usually labelled with labels such as NE, where NE refers to the upper right quadrant. As mentioned before, a quadtree is a data structure used for efficient manipulation of binary images. Informally, a quadtree is created by a recursive decomposition of a binary image I defined as follows. If the image I is homogeneous, i.e., either entirely black or white then the quadtree is a tree that consists of a single node with a label that corresponds to the binary value of the image, *black* or *white*. If the image I is nonhomogeneous, we decompose the image into four equal size quadrants $\{I_0, I_1, I_2, I_3\}$ and recursively compute the quadtrees for each $\{Q_0, Q_1, Q_2, Q_3\}$. The quadtree of I is a tree with root R connected to Q_i with an edge labelled i. The root is labelled with G. In this paper we address parallel algorithms for set theoretical operations on images represented by quad-

1049-9660/94 $6.00

trees. In the next section we define a fundamental operation on trees that allows us to compute set operations efficiently.

3. CORRESPONDENCE OF QUADTREES

In order to perform operations such as unions or intersections of images represented by quadtrees, we must determine the overlapping regions in the images, namely, determine the corresponding quadrants. In this section we describe a general approach that allows us to compute efficiently the correspondence of nodes in trees. Correspondence of nodes in two trees T_1 and T_2 is defined recursively as follows:

1. The roots of T_1 and T_2 correspond.
2. Two nodes correspond if they are both the ith child of two corresponding nodes.

Correspondence is a fundamental operation on trees and has been addressed in many papers (see [DK90] for references). In [DK90] we show how to compute correspondence in $O(\log N)$ time using $N/\log N$ processors on CRCW PRAM. Correspondence can also be computed in $O(\log^2 N)$ time using $N/\log^2 N$ processors on EREW PRAM. See details in [DK90]. Both algorithms achieve optimal speedup by the standard use of Brent's theorem. Here we review the algorithm for CRCW PRAM.

We first give an approximation of the parallel algorithm for computing correspondence of two quadtrees. The total size of both trees is assumed to be less than or equal to N. Both trees represent images of the same size. We assume that each tree is stored in a standard contiguous representation and each node has an additional pointer to its direct parent. We associate a processor with each of the nodes in the tree. At termination, each node will have a pointer to the corresponding node in the other tree if such a node exists. In Algorithm Correspondence we sketch a first approximation.

ALGORITHM 1. (Correspondence, first approximation).

Step 1. For each node compute the path, i.e., the sequence of edge labels from the root to the node (e.g., (111, 112, etc.) that determines a unique path-name name(i) associated with the node i.

Step 2. For each tree create an array indexed by path names and put the pointer to node i in the array location that corresponds to name(i).

Step 3. Two nodes correspond iff they have the same path name. Let name(i) be the path-name to node i. The processor associated with node i reads the contents of location name(i) of the array that corresponds to the other tree. Thus, each node finds its corresponding node in the other tree.

The key problem with the approach as described above is in Steps 1–2. Since the depth of the tree could be $O(N)$ we may need N bits to describe some paths, that is, 2^N locations in the array. The approach above only works for trees whose depth is logarithmic in the size of the tree since we need only log N bits to describe each path. For clarity, we first describe the algorithm that ignores this difficulty and subsequently add a step that resolves the problem.

Step 1 is computed using the standard technique of doubling. Let F(node) denote the link of a node to its parent and let Path(node) denote the current path computed to the ancestor. Initially, Path(node) contains the label of the edge to the parent, i.e., 1, 2, 3, 4. We associate a processor with each node and repeat the following simple sequence of steps. For each node in the tree:

```
1. Path(node) ← Concatenate ( Path(node), Path (F(node) ).
2. F(node) ← F(F(node).
```

Intuitively, each node computes a one-length path up, then paths of length 2, 4, and so on. In general, in parallel step i each node computes its path to its 2^i ancestor.

Steps 2 and 3 are constant time steps. Clearly, if we can index an array of size N by the paths computed in this fashion the algorithm terminates in $O(\log N)$ time. However, the depth of trees is not necessarily logarithmic in the size of the tree and the path descriptors quickly become too long. It is generally assumed in a RAM model of computation that we can operate in constant time only on log N bits. Below we describe a modification to the algorithm that solves the problem. This adds a factor of log N to the complexity of the algorithm on an EREW PRAM. However, it can be implemented on a CRCW in constant time. The simple technique for the CRCW PRAM model of computation has been used in this context in several previous papers [DK90, KP88].

The technique we use relies on the power of concurrent writes and is based on the following simple observation. We assume, inductively, that each path identifier is an integer in the range $[0, N]$ (this clearly holds initially). Therefore, after each doubling (concatenation) step of the algorithm in Step 1, there are at most N paths descriptors in the range $[0, N^2]$, i.e., each 2 log N bits long. Each path corresponds to a node in a tree. We must find a scheme to generate N unique identifiers in the range $[0, N^2]$ corresponding to these path names, such that two paths have the same name iff they correspond to the same path. However, paths described by 2 log N bits can be condensed into log N bit descriptors by a variety of methods. Here we describe one standard technique. We use a linear array of size $[0, N^2]$. Each processor writes its own processor-identifier in a location that corresponds to the path identifier the processor has currently computed.

One of the concurrent writes succeeds. Then each processor reads the location it attempted to write into. Each processor now has obtained a unique identifier (in the range $[0, N]$) for the path descriptors (in the range $[0, N^2]$). We note that the space complexity of this algorithm is quadratic but it can be brought down to almost linear by a variety of methods [MV90].

A different approach has been used by Ramesh *et al.* [RVKR87]. They observe that the conversion from path identifiers described by $2 \log N$ bits into path identifiers that use $\log N$ bits can be solved using sorting. Sorting can be performed in $O(\log)N$ time with N processors [Col86] on EREW. The sorting process will add $O(\log N)$ time complexity for the unique identifier generation process, which will slow down the algorithm by a factor of $O(\log N)$. Now we give the modified algorithm for correspondence in quadtrees.

ALGORITHM 2. (Correspondence, refined).

Step 1. Computing path sequences to nodes using doubling.

```
Repeat log N times:
    For each node in both trees:
    Path(node) = Concatenate ( Path(node),
      Path (F(node) ).
    F(node) = F(F(node).
    Compress all Path identifiers of length
      greater than log N.
```

Step 2. For each tree create an array indexed by path names and put the pointer to node i in the array location that corresponds to name(i).

Step 3. Two nodes correspond iff they have the same path name. Let name(i) be the path name to node i. The processor associated with node i reads the contents of location name(i) of the array that corresponds to the other tree. Thus, each node finds its corresponding node in the other tree.

Note that in Step 1 we must perform the path concatenation and compression in both trees simultaneously. Otherwise, it is generally not possible to give unique structural encoding to trees with N nodes using less than N bits per path. The algorithm described above gives us an $O(\log N)$ algorithm for computing corresponding nodes in quadtrees (assuming compression of path-identifiers can be done in constant time as outlined above). The number of processors used is N. Thus, the processor-time product is $O(N \log N)$ which is not optimal. The node correspondence algorithm as outlined in Algorithm 2 has been described in [RVKR87]. They implement the compression using sorting and therefore obtain a parallel algorithm that runs in $O(\log^2 N)$ time using N processors. We implement the compression in constant time using the power of concurrent writes. In the next section we describe a tree reduction technique to achieve optimal speedup. This technique has been first introduced in [DK90] in the context of term matching problems in artificial intelligence.

3.1. *Achieving Optimal Speedup*

In this section we show how to attain an optimal speedup. Given P processors, where $P \leq N/\log N$, we achieve N/P running time on an CRCW PRAM. The method is described in detail in [DK90]. Here we give an informal description. The idea is to preprocess the original trees P and G to reduce their sizes by a factor of $\log N$. We then apply the procedure described in the previous section to the reduced trees. Finally we add a postprocessing step to complete the correspondence process for all nodes in the original trees.

For simplicity assume the depth and size of the trees are 2^i for some $i \geqslant 1$. We start by marking every node u in the original trees P and G that has the following properties: (1) u is on level $k \log N$, for some positive integer k; and (2) the subtree rooted at u contains at least $\log N$ nodes. Such nodes can be determined with $O(N/\log N)$ processors in $O(\log N)$ time. Level information and the size of the tree below any node can be done using standard techniques. The reader should verify that the number of nodes marked by the above procedure is at most $N/\log N$.

We now generate trees P' and G' from P and G by removing all unmarked nodes and connecting the marked nodes as children of their nearest marked ancestors in the original tree. Clearly, the new trees, consisting of only the marked nodes are of size at most $N/\log N$. We can label each edge (u, v) in P' and G' with an integer label in the range $1 \cdots N$ as determined by the sequence of labels on the path connecting u and v in the original tree P or G. These labels can be computed on an $O(N/\log N)$-processor CRCW PRAM in $O(\log N)$ time by simply scanning the nodes on each level from top-to-bottom, since there are exactly $\log N$ levels involved. Now we compute the correspondence of the trees P' and G'. We utilize the algorithm we described in Algorithm 2 which computes the correspondence of two trees of size N in time $\log N$ with N processors. Since the trees are reduced to be of size at most $N/\log N$, our algorithm needs only $N/\log N$ processors.

After the correspondence of the reduced trees has been determined we can determine the correspondence of the nodes in the dropped subtrees by a simple top down parallel traversal of the subtrees from each one of the corresponding nodes. This can be done since the depth of trees that got dropped during the reduction step is at most $O(\log N)$. The implementation of this step can be done using standard algorithms such as list ranking and Euler tours (see [CV86] for details). The technique described above allows us to compute the correspondence of two trees in optimal log N time with $N/\log N$ processors.

In the next section we show how to apply the technique to compute intersection, union, and subtree operations on quadtrees.

4. UNIONS OF QUADTREES

The parallel algorithm for computing the union of two images represented by quadtrees T_1 and T_2 consists of three basic steps:

1. Identification of corresponding nodes.
2. An OR operation on the corresponding leaves of the two trees:
   (a) OR(*black*,anything) = *black*.
   (b) OR(anything,*black*) = *black*.
   (c) OR(*white*,*white*) = *white*.
3. Merging of black nodes.

The algorithm described in the previous section solves the node correspondence problem. Note that many nodes will not have corresponding nodes in the other tree. Once the corresponding nodes have been determined, we simply perform a local "OR" operation on the corresponding nodes. We mark all the nodes that have found a corresponding node in the other tree. Note that the marked subtrees in T_1 and T_2 are identical. We now create a new tree which is a copy of the marked subtrees. This is a constant step operation (in practice we can actually use one of the original subtrees to save storage). Each node in the new tree has two corresponding nodes in the input trees. The label associated with each node in the new tree is an OR of the labels associated with the two corresponding nodes in the input trees. Each node u in the new tree checks whether it is a leaf. This could occur iff one or both of the nodes, that u was generated from in the input trees, are leaves. If this occurs, then the node copies the pointers to the successors of the nonleaf node that it corresponds to (if both are leaves the new node becomes a leaf as well). The new tree is basically a union of the two trees, but it may contain unmerged nodes.

The only remaining work to be done is to merge the black nodes (step 3 above) in the new tree. This operation can be accomplished by a variety of methods. One simple method is to use the arithmetic expression evaluation technique. The arithmetic expression evaluation problem is a fundamental problem in parallel computing. Given an arithmetic expression represented as a tree where the internal nodes are arithmetic operators (e.g., addition) and the leafs are operands (e.g., integers), we compute a partial value for each node as determined by the arithmetic expression associated with the subtree that starts at this node.

We associate with each internal node an equation of the form:

value(node) = AND(child1, child2, child3, child4).

The AND of the four nodes is *black* if all of them are *black*, and it is *gray* otherwise. Since AND is an associative and distributive operation, we can view this problem as a special case of arithmetic operation evaluation on trees, which can be solved in $O(\log N)$ parallel time with $N/\log N$ processors (see [MR85, KD88]). After the completion of this operation we perform two operations.

1. If the value computed by the arithmetic expression is *black* replace the previous label with *black*.
2. Check for each node if it has a *black* parent. If it does, disconnect (discard) the subtree rooted at the node from its parent.

Other methods for merging *black* nodes are possible. The steps for computing a union of two quadtrees are summarized in Algorithm 3.

ALGORITHM 3. (Union of quadtrees).

Step 1. Call Correspondence to solve the node identification problem; that is, for each node find its corresponding node in the other tree.

Step 2. Compute an "OR" of the nodes in corresponding locations in the two arrays.

1. OR(*black*,anything) = *black*.
2. OR(anything,*black*) = *black*.
3. OR(*white*,*white*) = *white*.

Step 3. Using the results of Step 2 and copying the appropriate pointers we create a new tree which is an extended union of the two trees (but contains unmerged black nodes).

Step 4. Merge black nodes.

Step 5. For each node in the new tree, check whether the father is *black*. If the father is *black* disconnect the pointer to the node in the tree. The resulting tree is the union of the two quadtrees.

This completes the computation of union of two quadtrees. Intersection of two trees can be computed in a similar fashion by modifying the local operation. We can solve additional problems on quadtrees using the techniques described above. It is easy to use the tree correspondence algorithm to test whether two trees are identical, i.e., testing for equality. Equality testing can also be performed optimally in parallel using depth-first numbering and the degree of each node. Unions and equality can be used to perform other set operations. Consider the problem of checking whether an image X is a subimage of Y. Clearly, this can be checked by performing a union of X and Y and then checking whether the union is equal to Y.

5. DISCUSSION

To summarize, we have described several basic parallel quadtree algorithms. In the sequential model of computation the problems addressed in this paper can be solved optimally by tree traversal techniques. This is no longer true in the context of parallel computation. In this paper we tried to draw attention to the variety of parallel techniques that can be used to achieve asymptotically optimal performance for spatial data-structures. The algorithms utilize a wide range of fundamental parallel techniques such as parallel doubling, Euler tours, tree reduction, unique identifier generation, path compression, and arithmetic expression evaluation. These techniques were essential in achieving processor-time optimal algorithms. We established that node correspondence is the major bottleneck in the parallel computation of set operations on quadtrees. Once this correspondence has been computed, the remaining computations are relatively easy and can be accomplished using standard techniques. Additional techniques for tree correspondence can be found in [DK90]. There we describe $N/\log^2 N$ processor, $O(\log^2 N)$-time algorithms for a shared memory model that prohibits simultaneous reads and writes (EREW PRAM). The space complexity of the algorithms is linear in the size of the tree. The techniques are simple and can be implemented on a mesh used to achieve optimal algorithms for set operations on region quadtrees in optimal $O(\sqrt{N})$ time [DK89].

REFERENCES

[Bre74] R. P. Brent, The parallel evaluation of general arithmetic expressions, *J. Assoc. Comput. Mach.* **21,** 1974, 201–208.

[BRW88] A. S. Bhaskar, A. Rosenfeld, and A. Wu, Parallel processing of regions represented by linear quadtrees, *Comput. Vision Graphics Image Process.* **42,** 1988, 371–380.

[Col86] R. Cole, Parallel merge sort, in *Proceedings, 27th Symposium on the Foundations of Computer Science, 1986*, pp. 511–516.

[CV86] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70,** 1986.

[DK89] A. L. Delcher and S. Kasif, Term matching on a mesh-connected parallel computer, in *Israeli Symposium on Artificial Intelligence, Vision and Pattern Recognition, 1989.*

[DK90] A. L. Delcher and S. Kasif, Efficient parallel term matching, in *Proceedings, 1990 International Conf. on Logic Programming, 1990.*

[ES85] A. S. Edelman and E. Shapiro, Quadtrees in concurrent prolog, in *Proceedings, Int. Conf. Parallel Processing, 1985*, pp. 544–551.

[KD88] S. R. Kosaraju and A. L. Delcher, Optimal evaluation of tree-structured computations by raking, in *Proceedings, AWOC, 1988.*

[KP88] Z. M. Kedem and K. V. Palem, *Optimal Parallel Algorithms for Forest and Term Matching*, Technical report, IBM Thomas J. Watson Research Center, June 1988.

[MR85] G. L. Miller and J. Reif, Parallel tree contraction and its applications, in *Proceedings, 26th Symp. on Foundations of Computer Science, 1985.*

[MV90] Y. Matias and U. Vishkin, Parallel hashing and integer sorting, in *Proceedings, 17th ICALP, 1990*, pp. 729–743.

[RVKR87] R. Ramesh, R. M. Verma, T. Krishnaprasad, and I. V. Ramakrishnan, Term matching on parallel computers, in *Proceedings, 14th International Conference on ALP, 1987.*

[S84] H. Samet, The quadtree and related hierarchical data structures, *ACM Comput. Surveys* **16,** 1984, 187–260.