Construction of d-Dimensional Hyperoctrees on a Hypercube Multiprocessor *

Frank Dehne † **Andreas Fabri** ‡ **Mostafa Nassar** §
Andrew Rau-Chaplin † **Rada Valiveti** †

Abstract

We present a parallel algorithm for the construction of the hyperoctree representing a d-dimensional object from a set of n $(d-1)$-dimensional hyperoctrees, representing adjacent crossections of this object. On a p-processor SIMD hypercube the time complexity of our algorithm is $O(\frac{m}{p} \log p \log n)$, where m is the maximum of input and output size.

[1] *This work was partially supported by the Natural Sciences and Engineering Research Council of Canada and the ESPRIT Basic Research Actions Nr. 3075 (ALCOM) and Nr. 7141 (ALCOM II).*
[2] *School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6*
[3] *INRIA — B.P.93 — 06902 Sophia-Antipolis cedex, France*
[4] *Jodrey School of Computer Science, Acadia University, Wolfville, Canada B0P 1X0*

1 Introduction

In many applications a description of an object is given as a set of cross sections. For example in medical diagnosis, planar cross sections are obtained via tomography systems. In this paper we consider the problem of computing the 3-dimensional representation of an object given as such a set of cross sections. There exist two primary versions of this problem depending on what is known about the input. If the cross sections are known to be widely spaced or we have no spacing information at all, i.e. the information about the 3-dimensional object consists only of planar contours extracted from the cross sectional images, then we have an interpolation problem (see for example [Bo88]). Alternatively, if the cross sections are known to lie close together, i.e. they can be considered as interpolations of the volume lying between themselves and the following cross section, then we have a merging problem. In this paper we consider the latter version of the problem.

The representation of 2 and 3 dimensional objects has been extensively studied ([R80, M88, S89]). One commonly used representation scheme, based on recursive subdivision, is the *quadtree*. A binary image I is represented by a 4-ary tree consisting of *black*, *white*, and *grey* nodes. The root r of the tree represents the entire image I. If I is entirely black or white, the r is black or white, respectively, and has no children. Otherwise, r has four children recursively representing the four quadrants of I. We refer to [S84] for an overview and bibliography on quadtrees (incl. its many variants) and quadtree applications. The 3-dimensional and d-dimensional generalizations (for fixed constant d) are called *octrees* and *hyperoctrees*, respectively [YS83]. Since quadtree applications in image processing, solid modeling etc. are typically data intensive, the application of parallelism to such a fundamental data structure is of both theoretical and practical interest. While some papers ([MCI86, ML86]) consider parallel architectures designed (or reconfigured) particularly for quadtree manipulation, others consider general purpose architectures, such as PRAMs ([BRW88]), mesh-connected computers ([HR89]) and hypercubes ([DFR91, IK92]). Many of the construction and manipulation algorithms given in these papers can be easily adapted to work for d-dimensional hyperoctrees.

In this paper we present a parallel algorithm for the construction of the hyperoctree representing a d-dimensional object from a set of n $(d-1)$-dimensional hyperoctrees, representing adjacent crossections of this object. On a p-processor SIMD hypercube, the time complexity of our algorithm is $O(\frac{m}{p} \log p \log n)$ where m is the maximum of the input and output size.

Note that both the input and output hyperoctrees may be based either on a pointer based or on a linear representation (to be explained below). The sequential algorithm [YS83] has a running time of $O(m \log n)$. Our parallel algorithm follows the basic merging strategy of [YS83]. The main contribution of this paper is to solve the non trivial problem of merging "hybrid trees" in parallel.

The remainder is organized as follows. In the next section we describe our model of computation and some basic operations. In Section 3 we recall the definition of hyperoctrees and give a formal definition of a new data structure called hybrid trees. Section 4 briefly recalls the sequential algorithm and is followed by a section describing a merging algorithm for hybrid trees. Section 6 describes in detail our parallel hypercube algorithm.

2 The Model of Computation

In this section, we present two abstract models of a SIMD hypercube and some basic algorithms for this type of architecture.

A *SIMD hypercube* of dimension d consists of $p = 2^d$ processors which are indexed 0 through $2^d - 1$. Two processors are connected along dimension i, if and only if the binary representation of their indices differ in exactly the i^{th} bit. The processors are synchronized and may be enabled or disabled to execute a common instruction. Each processor has some local memory. Note that we neither assume a constant amount of memory per processor (as is done by exclusively fine-grained algorithms) nor a fixed non-constant amount of memory (as is assumed by exclusively coarse-grained algorithms). Our algorithms are suitable for implementation on either fine-grained or coarse-grained SIMD hypercubes and therefore could be implemented on machines ranging from Intel's iPSC/860 [In] (using additional synchronization) to Thinking Machines Corporation's CM2 [St87].

On constant size data, arithmetic operations on each processor and communication between processors which are adjacent along a fixed dimension take time $O(1)$.

In this paper we will use many *basic vector operations* such as parallel prefix, monotonic routing and bitonic merge [NS81, B68]. The parallel prefix sum of a vector V is the vector W, with $W[k] := \sum_{i=0}^{k} V[i]$, $0 \leq k < p$. Instead of summing we can perform any binary associative operation, e.g. copying. A further generalization is the segmented parallel prefix. The vector is split into segments and the parallel prefix starts at the beginning of each segment. In this paper we also use an operation called segmented broadcast, that is in each segment all elements are replaced by the first element in the segment.

Monotonic routing refers to the following operation. Given a data vector V, a destination vector D and a Boolean vector selecting some elements of V, the selected data elements $V[i]$ are moved to $V[D[i]]$ under the condition that $D[i] < D[j]$ for all selected elements $0 \leq i < j < p$. That is, the selected data elements remain in the same order.

The bitonic merge algorithm transforms a bitonic sequence (in our case, the keys are first increasing and then decreasing) into a sorted sequence. This operation can be used to merge two sorted sequences.

All these vector operations are presented for vectors of length p and they take time $O(\log p)$. They can easily be generalized for vectors of length n, $n \geq p$, and take then time $O(\frac{n}{p} \log p)$.

The second model under consideration is the *pipelined hypercube*. Arithmetic operations again take time $O(1)$, but processors can communicate with all adjacent processors in time $O(1)$. Thus we can pipeline the above basic vector operations and get $O(\frac{n}{p} + \log p)$, $n \geq p$, as time bounds. The CM2 from Thinking Machines Corporation is an example of such a machine [BLM*91].

3 Hyperoctrees and Hybrid Trees

The d-dimensional hyperoctree [YS83], or for short 2^d-tree, is a quadtree generalization for the representation of a d-dimensional discrete grid of sidelength n, where n is assumed to be a power of 2. Instead of splitting the 2-dimensional grid (image) into four quadrants, the d-diensional grid is split into 2^d subgrids of half the sidelength each. The subgrids are split recursively and recursion stops when subgrid of uniform colour are reached. Nodes are either grey, black or white, representing nonuniform, black or white subgrids. The *level* of a node v is defined recursively as follows: $\text{level}(v) := \log_2 n$ if v is the root, otherwise $\text{level}(v) := \text{level}(\text{father}(v)) - 1$.

For our algorithm we need a further generalization of 2^d-trees, since we want to represent discrete grids where the lengths of the sides in the d-th dimension vary.

To understand why this generalization is necessary consider the following naive algorithm for combining n 2^{d-1}-trees to form a 2^d-tree.

1. Convert each of the 2^{d-1}-trees that represents a slice into an equivalent

2^d-tree.

2. Perform a union operations on all the 2^d-trees formed in the previous step.

The problem with the naive algorithm is that it can result in a considerable and unnecessary "blow up" in the amount of data that must be processed. Consider for example the situation in which all of the $(d-1)$-dimensional slices are completely black. In the first step of the naive algorithm each of the fully black $(d-1)$-dimensional slices would be broken up in n^{d-1} volume elements of sidelength 1, as the width along dimension d is only 1. Thus we would return to a representation in image space. When in the second step of the naive algorithm the union of all n slices is computed the resulting 2^d-tree is of size $O(1)$. It is exactly this unnecessary increase or "blow up" in data size that any efficient algorithm must seek to avoid.

We need to represent not only d-dimensional cubes but also cuboids with the same sidelength k in dimensions 1 through $d-1$ and sidelength $j \leq k$ in dimension d, where j is the number of already joined slices, and k and j are powers of 2.

Definition 1 A *phase i hybrid tree T^i* is a colored tree with two kinds of nodes. The nodes at levels l, $i \leq l \leq \log n$, form a 2^{d-1}-tree and represent *cuboids* of size $(2^l)^{d-1} \times 2^i$, and the nodes at levels 0 through i form a forest of 2^d-trees and represent *cubes* of size $(2^l)^d$.

Notice that phase 0 or $\log n$ hybrid trees can be considered as 2^{d-1}-trees or 2^d-trees, respectively. Further note that nodes at level i are at the same time leaves of the 2^{d-1}-subtree and roots of the 2^d-subtrees. Figure 1 shows a phase $i = 2$ hybrid tree for $d = 2$. The nodes at level $l > 2$ represent rectangles of height $2^i = 4$ and length 2^l, nodes at levels $l \leq 2$ represent squares of sidelength 2^l.

4 Sequential Algorithm

The above definition of hybrid trees formalizes an intermediate state of the sequential algorithm presented in [YS83]. The basic structure of their algorithm is to consider each 2^{d-1}-tree as a phase 0 hybrid tree and perform $\log n$ pairwise merge phases. More precisely, in phase $i+1$ of the merge algorithm, $0 \leq i < logn$, pairs T_1^i and T_2^i of phase i hybrid trees are merged into a phase $i+1$ hybrid tree $T_{1,2}^{i+1}$.

Figure 1: A hybrid tree

In the following section, we present a modified version of this merge phase, which we will use in our parallel algorithm.

5 Merging Hybrid Trees

We describe an algorithm to join two hybrid trees T_1^i and T_2^i in phase $i+1$ of the merge. For this algorithm to be efficient it must avoid performing work that will later have to be undone. In particular, the decision to break up a cuboid should be postponed as long as possible, i.e. until it is certain that the cuboid will never become part of a uniformly coloured cube. When a cuboid is broken up, it must be broken up into cuboids which are maximal and which also fit this criterion.

Before we can join the trees we overlay the 2^{d-1}-subtrees of T_1^i and T_2^i and expand them to the same shape. While expanding the trees, we consider three different cases for all corresponding nodes v_1 and v_2 at levels $l > i$ in the two trees.

Case 1: If v_1 is black or white and v_2 is grey, then the cuboid of v_1 must be broken up to the same level of resolution as the cuboid of v_2. Therefore we replace v_1 with a copy of the 2^{d-1}-subtree rooted at v_2, where leaf nodes get the colour of v_1. We do not have to copy nodes below level i, as nodes in level i already represent cubes.

Case 2: If v_1 is black and v_2 white, they must both be broken up into cubes of sidelength 2^i. We thus replace each of them with the complete 2^{d-1}-tree of height $l - i$ with leaf nodes coloured as v_1 and v_2, respectively.

Figure 2: Case 1 for $d = 2$

Case 3: If they are both either black or white, they represent cuboids of the same length and form together a cuboid of height 2^{i+1}. Thus, no expansion is necessary.

The 2^{d-1}-subtrees of the two trees are now of the same shape and corresponding nodes on levels $l \geq i+1$ have the same colour. We use the overlayed trees to construct the 2^d-tree nodes on level $i+1$ by joining the children of corresponding grey 2^{d-1}-tree nodes at level $i+1$. Keeping only one copy of the expanded 2^{d-1}-trees we obtain the hybrid tree $T_{1,2}^{i+1}$.

On a sequential machine the above merge operation can be easily implemented in time $O(\max(|T_1^i| + |T_2^i|, |T_{1,2}^{i+1}|))$ by performing simultaneous tree traversals [YS83]. As the overall algorithm consists of $\log n$ phases, the input set of n $(d-1)$-dimensional hyperoctrees can be transformed into the corresponding d-dimensional hyperoctrees T in time $O(m \log n)$, with $m = \max(|T|, \sum_{j=1}^{n} |T_j|)$.

Figure 3: Case 2 for $d = 2$

6 Parallel Algorithm

In the parallel setting, an efficient implementation of hybrid tree merge is not as straightforward. Since the sequential time complexity is bounded below by $\Omega(m)$, an optimal parallel algorithm, even for the PRAM model, can not achieve a time better than $O(\frac{m}{p})$. In the following, we present an $O(\frac{m \log n}{p} \log p)$ time hypercube algorithm.

We will use a linear representation instead of the pointer based representation of hybrid and 2^d-trees used in the sequential setting. A *linear tree* is a collection of just the leaf nodes, ordered according to the inorder traversal of the pointer based tree. We will later show how to convert between this linear representation and the level-order pointer-based representation described in [DFR91].

Consider a 2^{d-1}-subtree T' of a hybrid tree and a node v in T'. Let $\bar{T}$ be the expansion of T' to a complete $(d-1)$-ary tree of height $\log n$, and $\bar{v}$ be the corresponding node of v in $\bar{T}$. We define the position of v, $pos(v)$, to be the inorder traversal number of $\bar{v}$ in $\bar{T}$. Let $cover(v)$ be the interval $[lm(v), rm(v)]$, where $lm(v)$ and $rm(v)$ are the inorder traversal numbers (with respect to the complete tree $\bar{T}$) of the leftmost and rightmost descendants (in $\bar{T}$) of $\bar{v}$. For each node in a 2^d-subtree we define $anc_i(v)$ to be the position of the ancestor of v in level i. Note that the size of these numbers is at most $(d-1)\log n$ bits.

We now show how to merge two hybrid trees T_1^i and T_2^i to perform phase $i+1$ of the algorithm. We assume w.l.o.g. that the slice represented by T_1^i lies above the one represented by T_2^i. Let $seq(T_1^i)$ and $seq(T_2^i)$ be the node sequences of the linear tree representations of T_1^i and T_2^i, respectively. The following describes the parallel implementation of cases 1, 2 and 3 from Section 3. In our algorithm, the expansion step and the join step are not performed separately.

Case 1: In this case each black or white node, v_1, in T_1^i corresponding to a grey node, v_2, in T_2^i is replaced by a tree of the shape of T_{v_2}, the 2^{d-1}-subtree of the tree T_2^i rooted at v_2. As internal nodes are not represented by the linear representation, this is equivalent to replacing v_1 by the leaves of T_{v_2}. We perform this operation in two steps.

In the *first step* we detect every black or white node v_1 in levels $l > i$ of T_1^i, which corresponds to grey node v_2 of T_2^i. Such a black or white node v_1 is called a *covering node*. The leaves of the subtree of T_2^i rooted at v_2 are called *covered* by v_1. All covering nodes are detected as follows: We merge the tree sequences $seq(T_1^i)$ and $seq(T_2^i)$ by assigning $pos(v)$ as the key to each node in a level $l > i$ and $anc_i(v)$ as key to each node in a level $l \leq i$. Let π be the resulting sorted sequence. We observe the following. A node v_1 in a level $l > i$ is a covering node, if and only if one of its neighbours in π has a different *pos* value and is contained in the interval $cover(v_1)$. Thus, every node performs a comparison with its two neighbours. We then unmerge the sequence π, i.e. extract the two original sequences $seq(T_1^i)$ and $seq(T_2^i)$.

In the *second step* we determine for each covering node v_1 the set of leaves of the subtree T_{v_2} rooted at the corresponding node v_2. We first create a copy of each covering node and store it at the same processor. Every node v in T_1^i or T_2^i is then assigned a key, $key(v) = (x, y)$, where x and y are the primary and secondary keys, respectively. We define $key(v)$ as follows:

$$key(v) = \begin{cases} (lm(v), 0) & \text{if } v \text{ is the } 1^{st} \text{ copy of a covering node at level } l > i, \\ (rm(v), 3) & \text{if } v \text{ is the } 2^{nd} \text{ copy of a covering node at level } l > i, \\ (pos(v), j) & \text{if } v \text{ is a noncovering node in } T_j^i,\ j = 1, 2, \text{ at level } l > i, \\ (anc_{i+1}(v), j) & \text{if } v \text{ is a node in } T_j^i,\ j\text{=1,2, at level } l \leq i. \end{cases}$$

Lemma 1 *The nodes in the linear hybrid trees form a monotonic sequence with respect to $key(v)$.*

Proof: The nodes are ordered by their *pos* value. As only leaf nodes are represented, all $lm(v)$ and $rm(v)$ values are unique and no *pos* value of another node is contained in the interval $cover(v)$. Noncovering nodes at level $l > i$ do not change their key. Let $(u_1 \dots u_k)$ be a maximal subsequence of nodes of the linear tree T_j, which are at level $l \leq i$ and receive the same key. Let w be their ancestor in level $i + 1$, then $pos(u_1) < pos(w) < pos(u_k)$.
□

Thus, we can join the two trees using a stable bitonic merge. The following lemma states some observations on the resulting sequence S. We omit the proof of the statements as they are a direct consequence of the choice of $key(v)$.

Lemma 2 *The sequence S has the following properties:*
(a) *The first copy of a covering node v appears immediately to the left of the first node of the subsequence of nodes covered by v, the second copy immediately to the right of the last node of the subsequence of nodes covered by v.*
(b) *The 2^d-trees which become children of a node in level $i+1$ of the resulting hybrid tree $T_{1,2}^{i+1}$ are in the order required for $T_{1,2}^{i+1}$.*

(c) *Corresponding black or white 2^{d-1}-tree nodes of T_1^i and T_2^i, respectively, are neighbours in S.*

We use Lemma 2(a) to compute for all covered nodes their respective covering nodes. We do so by performing a segmented broadcast of covering nodes, where the segments are defined by the two copies of the covering nodes. As *cover* intervals do not overlap, any node is covered by at most one node. Then both copies of covering nodes are removed from S.

We now create the copies of the leaves of T_{v_2}, the 2^{d-1}-subtree rooted at v_2, for the covering node v_1. Each covered subsequence $\hat{S}$ of 2^d-tree nodes with the same key forms the set of leaves of a subtree rooted in level $i+1$. We thus insert 2^{d-1} copies of the covering node before or behind $\hat{S}$, depending on whether the covering node comes from T_1^i or T_2^i.

Case 2: We expand corresponding nodes v_1 and v_2 where one of the nodes is black the other node white. According to Lemma 2(c) corresponding nodes can be determined by a comparison with the neighbour in the sequence S. Let the two corresponding nodes be at level l. Instead of replacing them first by two 2^{d-1}-trees which are then joined in a second step, we directly construct the 2^d-tree. To achieve this we replace them by 2^{i-l} groups of 2^{d-1} nodes at level i. The groups alternate their colours as v_1 and v_2 do.

Case 3: The removal of one copy of corresponding black or white nodes completes the joining of two 2^{d-1}-trees and yields the tree $T_{1,2}^{i+1}$.

This concludes our discussion of a phase $i+1$ merge of two trees T_1^i and T_2^i. As we start with n hyperoctrees $T_1, \ldots, T_n$, we need $\log n$ such phases to obtain the final 2^d-tree, T. The time we need to perform one phase depends on the input size as well as on the output size. This can be illustrated with the following examples. If all slices are black, we start with n single node trees and end up with one node representing a black block. On the other hand, if slices are alternating black and white, we start with n single node trees and end up with one tree of size n^d. As the algorithm only constructs nodes which either appear in the final output tree T or which are further refined, we can bound the data size by $m = \max(|T|, \sum_{i=1}^{n} |T_i|)$.

The processor load may change in each phase, and the insertion and deletion of nodes is in fact a reallocation of processors. This can be performed with parallel prefix and monotonic routing operations. Since the nodes always remain in the same order it requires time $O(\frac{m}{p} \log p)$.

In a preprocessing step we compute the keys *pos* with local operations in time $O(\frac{m}{p} \log m)$. For each node v we start with the position of the root and follow the path from the root to v. It takes time $O(1)$ to compute the position of a child of node v, given the position of v. The height of the tree is $O(\log m) = O(d \log n) = O(\log n)$.

Given *pos*, the value of *lm* and *rm* can be computed in time $O(1)$ using closed formulas as the tree is a complete tree.

In each phase we have to compute keys for nodes which were either inserted or which changed from the 2^{d-1}-subtree to a 2^d-subtree. Nodes which are inserted obtain their keys from the nodes which caused their insertion. Given anc_i, the value of anc_{i+1} can be computed in time $O(1)$.

This results in the following theorem.

Theorem 1 *The construction of a linear d-dimensional hyperoctree from a set of n linear $(d-1)$-dimensional hyperoctrees on a hypercube with p processors takes time $O(\frac{m \log n}{p} \log p)$, where m is the maximum of input and output size.*

We now describe how to construct a d-dimensional *pointer-based hyperoctree* from a set of n $(d-1)$-dimensional pointer-based hyperoctrees. The pointer- based representation is at times an attractive alternative to the linear representation, even in the parallel setting [DFR91]. Our approach will be to convert the given pointer-based representation into a linear representation, apply the algorithm given above to perform the merging, and then convert back to a pointer-based representation. We will assume that the n $(d-1)$-dimensional pointer-based trees are given in level-order. That is, for each tree the nodes are ordered by hight, and nodes with the same hight are ordered "left-to-right". For more details see [DFR91].

To convert a pointer-based tree T_p into its equivalent linear representation T_l, all we need to compute is the inorder number of each node v in T_p and then sort the leaves of T_p by this value. The resulting sequence of nodes is T_l.

To convert a linear tree T_l into its equivalent pointer-based representation T_p, we start by computing for each node v of T_l the values *pos(v)* and $anc_{i+1}(v)$ as previously described. The tree T_p is constructed level-by-level starting with the leaves. To form each level i of T_p we concentrate all level i nodes of T_l, and identify segments (groups) of nodes within this set having the same parents (i.e. $anc_{i+1}(v)$). The first node in each such segment then creates its parent node, together with their *pos* and *anc* values. Finally, the newly created parent nodes are merged with the nodes of T_l at level $i+1$ (and should be considered in the construction of the next level).

Clearly, both conversion algorithms run in time $O(\frac{n \log n}{p} \log p)$ as they consist of $\log n$ phases each requiring at most $O(\frac{n}{p} \log p)$ time. We can therefore state the following corollary to Theorem 1.

Corollary 1 *The construction of a pointer-based d-dimensional hyperoctree from a set of n pointer-based $(d-1)$-dimensional hyperoctrees on a hypercube with p processors takes time $O(\frac{m \log n}{p} \log p)$, where m is the maximum of input and output size.*

Given that the merging algorithm and the two conversion algorithms described above use only basic hypercube operations that can be pipelined, we immediately get the following corollary for both linear and pointer- based representations.

Corollary 2 *The construction of a d-dimensional hyperoctree from a set of n $(d-1)$-dimensional hyperoctrees on a pipelined hypercube with p processors takes time $O(\frac{m \log n}{p} + \log p \log n)$, where m is the maximum of input and output size.*

7 Conclusion

In this paper we presented a parallel algorithm for the construction of a d-dimensional hyperoctree from a set of n $(d-1)$-dimensional hyperoctrees, representing adjacent "slices". On a p-processor SIMD hypercube the time complexity of our algorithm is $O(\frac{m \log n}{p} \log p)$, where m is the maximum of input and output size. This parallel algorithm represents, to our knowledge, the first parallel algorithm for the construction of octree based data structures from anything but binary images or chain-codes.

References

[B68] K.E. Batcher. *Sorting networks and their applications.* Proc. AFIPS Spring Joint Computer Conference, pp. 307–314, 1968.

[BRW88] S. K. Bhaskar, A. Rosenfeld, and A. Y. Wu. *Parallel processing of regions represented by linear quadtrees.* Computer Vision, Graphics, and Image Processing, Vol. 42, pp. 371–380, 1988.

[Bo88] J.D. Boissonnat. *Shape Reconstruction from Planar Cross Sections.* Computer Vision, Graphics and Image Processing 44, pp. 1–29, 1988.

[DFR91] F. Dehne, A. Ferreira, and A. Rau-Chaplin. *Parallel processing of pointer based quadtrees.* Proc. International Conference on Parallel Processing, 1991.

[HR89] Y. Hung and A. Rosenfeld. *Parallel processing of linear quadtrees on a mesh-connected computer.* Journal of Parallel and Distributed Computing, Vol. 7, pp. 1–27, 1989.

[IK92] O.H. Ibbarra and M.H. Kim *Quadtree Building Algorithms on an SIMD Hypercube.* Proc. Int. Parallel Processing Symp. pp. 22–27, March 1992.

[In] Intel Scientific Computers *iPSC System Overview.*

[M88] M. Mäntylä. *An Introduction to Solid Modeling.* Comp. Sci. Press, 1988.

[MCI86] M. Martin, D. M. Chiarulli, and S. S. Iyengar. *Parallel processing of quadtrees on a horizontally reconfigurable architecture computing system.* Proc. Int. Conf. on Parallel Processing, pp. 895–902, 1986.

[ML86] G.-G. Mei and W. Liu. *Parallel processing for quadtree problems.* Proc. Int. Conf. on Parallel Processing, pp. 452–454, 1986.

[NS81] D. Nassimi and S. Sahni. *Data broadcasting in SIMD computers.* IEEE Trans. on Computers 30:2, pp. 101–106, 1981.

[R80] A. G. Requicha. *Representation for Rigid Solids: Theory, Methods and Systems.* ACM Computing Surveys, Vol. 12, No. 4, 1980.

[S84] H. Samet. *The quadtree and related hierarchical data structures.* Computing Surveys, Vol. 16, No. 2, pp. 187–260, 1984.

[S89] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1989.

[St87] C. Stanfill. *Communications Architecture in the Connection Machine System.* Thinking Machines Corporation — TR HA87-3, 1987.

[BLM*91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, M. Zagha. *A Comparison of Sorting Algorithms for the Connection Machine CM-2.* ACM Symp. on parallel algorithms and architectues, pp. 3-16, 1991.

[YS83] M. Yau and S. N. Srihari. *A Hierarchical Data Structure for Multidimensional Digital Images.* Comm. of the ACM, No. 1, Vol. 26, July 1983.

Figure 4: Merging linear trees T_1 and T_2 (thick black lines). In the first merge v_1 ends up between covered nodes. In the second merge its two copies deliniate the start and end of the sequence of covered nodes. Further the trees u_1, T_1', u_2 and T_2' become neighbours and form thus a linear quadtree rooted in level $i + 1$.