T U M

INSTITUT FÜR INFORMATIK

A New Efficient Algorithm for Embedding an Arbitrary Binary Tree into Its Optimal Hypercube

Volker Heun Ernst W. Mayr

TUM-I9321
August 1993

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-08-1993-I9321-100/1.-FI

Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

A New Efficient Algorithm for Embedding an Arbitrary Binary Tree into Its Optimal Hypercube

Volker Heun* Ernst W. Mayr *
Institut für Informatik
Technische Universität München
Arcisstr. 21, D-80290 München
Germany
{heun|mayr}@informatik.tu-muenchen.de

December 21, 1993

Abstract

The d-dimensional binary hypercube is a very popular model of parallel computation. On the other hand, the execution of many algorithms can be represented by binary trees, making desirable fast simulations of binary trees on hypercubes. In this paper, we present a simple one-to-one embedding of arbitrary binary trees into their optimal hypercubes with dilation 8 and constant congestion. The novelty of our method is based on the use of an intermediate quadtree data structure, which also permits the embedding to be efficiently computed on the hypercube itself. □

Topics: Algorithms and Data Structures, Theory of Parallel and Distributed Computation.

1 Introduction

Hypercubes are a very popular model for parallel computation because of their regularity and the relatively small number of interprocessor connections. On the other hand, the control structure and execution of many algorithms is represented by binary trees, e.g., in divide-and-conquer algorithms. Thus, it is desirable to find 'good' embeddings of binary trees into hypercubes to obtain efficient simulations.

It is well known that a complete binary tree has an embedding into a hypercube with dilation 1 and expansion 2, or with dilation 2 and expansion 1 (in the latter case, only one edge requires dilation 2). In terms of lower bounds, a simple parity argument shows that the complete binary tree of size 2^d-1 cannot be embedded into the d-dimensional hypercube with dilation 1 [SS85].

*Most of this work was done while the authors were members of the Fachbereich Informatik of the Johann Wolfgang Goethe-University Frankfurt (Main).

There exists an optimal hypercube algorithm for computing the embedding of a complete binary tree into its optimal hypercube. There even is a hypercube algorithm for embedding a complete binary tree into its optimal hypercube level by level, spending only constant time for each level [FM87].

For arbitrary binary trees, one-to-one embeddings into their optimal hypercubes with constant dilation and congestion have been constructed in [BCLR86, BCLR92, MS88].

In this paper, we present another, simpler algorithm for embedding arbitrary binary trees into their optimal hypercube. Furthermore, this embedding can be efficiently computed on the hypercube itself.

2 Preliminaries

An *embedding* of a graph $G{=}(V_G, E_G)$ into a graph $H{=}(V_H, E_H)$ is a mapping $\phi{:}G{\to}H$ consisting of two mappings $\phi_V{:}V_G{\to}V_H$ and $\phi_E{:}E_G{\to}\mathcal{P}(H)$. Here, $\mathcal{P}(G)$ denotes the set of paths in a graph $G{=}(V, E)$. The mapping ϕ_E maps each edge $\{v, w\}{\in}E_G$ to a path $p{\in}\mathcal{P}(H)$ connecting $\phi_V(v)$ and $\phi_V(w)$. The graph G is called *guest graph* and the graph H *host graph*. We call an embedding *one-to-one* if the mapping ϕ_V is 1-1.

The *dilation* of an edge $e{\in}E_G$ under an embedding ϕ is the length of the path $\phi_E(e)$. Here, the length of a path p is the number of its edges. The *dilation of an embedding* ϕ, denoted dil(ϕ), is the maximal dilation of an edge in G.

The number of vertices of a guest graph which are mapped onto a vertex v in the host graph, is called the *load* of the vertex v. The *load of an embedding* ϕ, denoted load(ϕ), is the maximal load of a vertex in the host graph. In this paper, unless noted otherwise, we only consider embeddings with load one. The ratio exp$(\phi){=}\frac{|V_H|}{|V_G|}$ is called the *expansion* of the embedding ϕ.

The *congestion* of an edge $e'{\in}E_H$ is the number of paths in $\{\phi_E(e) \mid e{\in}E_G\}$, containing e'. The *edge-congestion* is the maximal congestion over all edges in H. The *congestion* of a vertex $v{\in}V_H$ is the number of paths in $\{\phi_E(e) \mid e{\in}E_G\}$, containing v. The *node-congestion* is again the maximal congestion over all vertices in H.

A *hypercube of dimension* d is a graph with 2^d vertices, labeled 1-1 with the strings in $\{0, 1\}^d$. Two vertices are connected iff their labels differ in exactly one position. The smallest hypercube into which we can embed a graph $G{=}(V, E)$ with load one is called its *optimal* hypercube. The dimension of the optimal hypercube is $d{:=}\lceil\log(|V|)\rceil$[1)]. Thus, an embedding of a graph G into its optimal hypercube has expansion less than two.

In the following, we initially restrict our attention to finding a suitable mapping ϕ_V, and we will use shortest paths in the hypercube for the mapping ϕ_E. Nevertheless, it is still important to decide which paths we choose, since we are interested in obtaining an embedding with constant congestion.

To construct our embedding, we use the data structure of a weighted quadtree. The *weighted quadtree of height* h is a complete quadtree of height h with integer node weights, also called the *capacities* of the nodes. Here, the *height* of a tree is the number of vertices on a longest path from the root to a leaf. We distinguish two types of a

[1)] All logarithms in this paper are to the base 2.

weighted quadtree, which we call the *full* weighted quadtree and the *partial* weighted quadtree. The only difference is in the capacities. The root of the full (resp., partial) weighted quadtree has a capacity of $2^{k+1}{\cdot}4h$ (resp., $2^{k+1}(4h{-}2)$), and every other node at level l has capacity $2^{k+1}(3(h{-}l{+}1){-}1)$ (resp., $2^{k+1}(3(h{-}l{+}1){-}\frac{5}{2})$). Here, the *level* of a vertex is the number of vertices on the path from the root to the vertex. k is another integer parameter called the *order* of the weighted quadtree. In the following, we call vertices of a weighted quadtree *nodes*, and we denote the capacity of a node at level l by $c(l)$.

An easy computation shows that the total capacity in the full (resp., partial) weighted quadtree of height h and order k equals 2^{2h+k+1} (resp., 2^{2h+k}). Since we will use the weighted quadtree as an intermediate data structure for our embedding, and since the weight of a node will be equal to the number of vertices of the binary tree mapped to it, we will use a full (resp., partial) weighted quadtree, if $d{+}k$ is odd (resp., even). Thus, $d{=}2h{+}k{+}1$ if we use a full weighted quadtree, and $d{=}2h{+}k$ otherwise.

We now describe a mapping of a weighted quadtree into its optimal hypercube such that each node of the weighted quadtree occupies as many vertices of the hypercube as given by its capacity. Each node in a weighted quadtree can be represented by a string in $\{0,1\}^*$ of even length as follows. The empty string ε represents the root of the weighted quadtree. If α represents a node v, then the strings $\alpha00$, $\alpha01$, $\alpha10$, and $\alpha11$ represent the children of v from left to right. Note that the string representing a node at level l has length $2l{-}2$. We define the following sets of hypercube locations, where α represents some arbitrary node in a weighted quadtree:

$$\begin{aligned} S_\alpha &:= \left\{\alpha\beta1\gamma\delta \in \{0,1\}^d \,\middle|\, \beta\in\{\varepsilon,0,1\} \wedge \gamma \in 0^*10^* \wedge \delta\in\{0,1\}^k\right\} \\ S'_\alpha &:= \left\{\alpha1\gamma\delta \in \{0,1\}^d \,\middle|\, \gamma \in 0^*10^* \wedge \delta\in\{0,1\}^k\right\} \subset S_\alpha \\ T &:= \left\{\gamma\delta \in \{0,1\}^d \,\middle|\, \gamma \in (0^*10^*{+}0^*) \wedge \delta\in\{0,1\}^k\right\} \end{aligned}$$

We also define the set $S{:=}\bigcup_\alpha S_\alpha$. The vertices of the binary tree mapped to the node of a weighted quadtree represented by α, will finally be mapped to hypercube locations in the set $L_\alpha{:=}S_\alpha$ if $\alpha{\neq}\varepsilon$, and $L_\varepsilon{:=}S_\varepsilon \cup T$ otherwise. Note that for a leaf of a partial weighted quadtree represented by α, we have $L_\alpha{=}S'_\alpha{=}\left\{\alpha11\delta \,\middle|\, \delta{\in}\{0,1\}^k\right\}$.

It is easy to see that the capacity of a node in the weighted quadtree equals the cardinality of the set of vertices in the hypercube to which it is mapped. Furthermore, it can easily be verified that $S_\alpha \cap S_{\alpha'}{=}\emptyset$ and $S_\alpha \cap T{=}\emptyset$ for every $\alpha \neq \alpha'{\in}(\{0,1\}^2)^*$, and hence $L_\alpha \cap L_{\alpha'}{=}\emptyset$ for every $\alpha \neq \alpha'{\in}(\{0,1\}^2)^*$. Hence, for each string $s{\in}S$ there is a unique decomposition $s{=}\alpha\beta1\gamma\delta$ as used in the definition of S_α.

Lemma 2.1 *Let v and w be two nodes in a weighted quadtree such that their lowest common ancestor is at distance $\leq h'$ from both v and w. The distance of any pair of corresponding vertices in the hypercube to which v and w are mapped is at most $2h'{+}k{+}2$.*

Proof: We first consider the case where both vertices belong to the set S. The following diagram shows the labels of the two hypercube vertices to which v and w are mapped.

v:	α	α'	β'	1	γ'			δ'
w:	α	α''			β''	1	γ''	δ''

In this picture, α represents the root of the subtree and $\alpha\alpha'$ (resp., $\alpha\alpha''$) represents the vertex v (resp., w). Thus, $|\alpha''|\leq 2h'-2$ if we assume $|\alpha'|\leq|\alpha''|$. The definition of the mapping from the weighted quadtree to the hypercube implies that $|\beta''|\leq 1$, $|\delta''|=k$, and that γ' and γ'' contain exactly one 1 each. Hence the labels of the two hypercube vertices differ in at most $(2h'-2)+1+1+2+k=2h'+k+2$ positions.

We now consider the case that the vertex corresponding to v belongs to the set T implying that v in the image of the root of the weighted quadtree. The vertex corresponding to w, labeled $\alpha'\beta'1\gamma'\delta'$, belongs again to S. Since the height of the subtree is h', we have $|\alpha'|\leq 2h'-2$. Hence there are at most $2h'-2+1+1+1=2h'+1$ 1's in the first $d-k$ positions in the label of the hypercube vertex corresponding to w. By the definition of the set T, the label of the hypercube vertex corresponding to v has at most one 1 in the first $d-k$ positions. Thus, the labels of these two hypercube locations differ in at most $2h'+1+1+k$ positions.

Finally, if both vertices belong to the set T, their labels obviously differ in at most $k+2\leq 2h'+k$ positions. ∎

3 An Algorithm for Embedding Trees into Hypercubes

Our embedding of binary trees into hypercubes is achieved in two steps. First, we embed the binary tree into a weighted quadtree. Then, we use the mapping presented in the previous section to complete the embedding. To obtain a small dilation, adjacent vertices of the binary tree are mapped to locations which are close in the weighted quadtree. Our goal is to obtain an embedding of the binary tree into a weighted quadtree such that adjacent tree vertices are mapped to two nodes of the weighted quadtree with distance ≤ 2 from their lowest common ancestor of the weighted quadtree. We will show that this goal can be reached, except for tree vertices that get mapped to weighted quadtree nodes close to the leaves, where we obtain distance ≤ 3. Our method leads to an embedding of the binary tree into the hypercube with dilation $8+k$. The order k can be chosen to be 2. Using local modifications of the mapping, we can then reduce the dilation to 8.

3.1 Partitioning Forests

In this section we present some fundamental facts about decomposing forests. We start with a lemma on partitioning strings.

Let $s\in\{1,\ldots,m\}^*$. By $\sharp_i s$ we denote the number of occurrences of the symbol i in s, and by $\sharp s$ we denote the length of the string s. The following bisection lemma can be found in [GW85, AW86, BL84, C90].

Lemma 3.1 *Let $s\in\{1,\ldots,m\}^*$. There exists a decomposition of s into substrings s_i, for $i=1,\ldots,m+1$, such that $s=s_1\cdots s_{m+1}$, and a partition of $\{1,\ldots,m+1\}$ into sets J_1 and J_2, such that $\lfloor\frac{\sharp_k s}{2}\rfloor\leq\sum_{j\in J_\ell}\sharp_k s_j\leq\lceil\frac{\sharp_k s}{2}\rceil$ and $\lfloor\frac{\sharp s}{2}\rfloor\leq\sum_{j\in J_\ell}\sharp s_j\leq\lceil\frac{\sharp s}{2}\rceil$ for $k=1,\ldots,m$ and $\ell=1,2$.*

Proof: For lack of space we only give the proof for $m=2$ (which is all we use). We assume that n is even, a similar argument holds if n is odd. Without loss of generality, the number of 1's in the left half of s is at least as large as the number of 1's in the right half. Sliding a window of length $\frac{n}{2}$ across s, there must exist a position of the window such that the number of 1's in the window is $\lfloor \frac{\sharp_1 s}{2} \rfloor$. The number of 2's in the window then is $\frac{n}{2} - \lfloor \frac{\sharp_1 s}{2} \rfloor = \frac{\sharp_1 s + \sharp_2 s}{2} - \lfloor \frac{\sharp_1 s}{2} \rfloor = \lceil \frac{\sharp_2 s}{2} \rceil$. The two endpoints of the window give the required cuts for the strings. ■

We will need this proof later to find such a bisection of a string s. Note that Lemma 3.1 is optimal in the sense that there exist strings that cannot be divided as required using fewer substrings.

In order to partition forests of binary trees using the above partition lemma, we need a linear representation of such forests like inorder strings. Since we are interested in partitioning the forest cutting at most a logarithmic number of edges, and since the tree may be highly imbalanced, we cannot simply split the vertices into two parts as given by the lower and upper half of the inorder numbers. Instead we have to reorganize the trees in the forests. In the following we restrict our discussion to a single tree.

A tree vertex is called a *left (resp., right) vertex*, if it is the left (resp., right) child of its parent. For simplicity, the root is a left vertex. An edge is called a *left (resp., right) edge*, if it is the edge from a vertex to its left (resp., right) child.

We reorganize the binary tree in such a way that for any left (resp., right) vertex the size of its left (resp., right) subtree is at least as big as the size of its right (resp., left) subtree. In the remainder of this paper, we refer to this procedure simply as the *reorganization*. An efficient implementation of the reorganization on the hypercube is given in Section 4.1.

After the reorganization, a path from the root to a leaf can use at most $\lfloor \log(n) \rfloor$-times an edge from a left vertex to a right vertex or from a right vertex to a left vertex, since each such edge at least halves the number of vertices in the remaining subtree.

Note that cutting the string listing the vertices of the tree in inorder corresponds to cutting some edges of the path from the root to a leaf, and note that these edges alternate between right and left. In the following a cut edge is called a *partition edge*. Vertices incident to a partition edge are called *partition vertices*.

This construction together with Lemma 3.1 can be used to partition a forest of binary trees containing marked vertices into four forests. This yields the following lemma:

Lemma 3.2 *Let $F=(V,E)$ be a forest of binary trees containing marked vertices, and let V' be the set of marked vertices. Removing at most $6\lfloor \log(|V|) \rfloor - 2$ edges, F can be decomposed into four forests F_1, F_2, F_3, and F_4 such that the size of each forest is $\lfloor \frac{|V|}{4} \rfloor$ or $\lceil \frac{|V|}{4} \rceil$. Furthermore, the number of partition vertices and marked vertices in each F_k is at most $\lceil \frac{|V'|}{4} \rceil + 3\lfloor \log(|V|) \rfloor$.*

Proof: First, we apply Lemma 3.1 for $m=2$ to the inorder string of the reorganized forest. After the bisection, we mark all unmarked partition vertices and reorganize both newly created forests. Then, we apply Lemma 3.1 for $m=2$ once more to the inorder string of each forest. Finally, we remove the marks from the unmarked partition vertices of the first bisection. For an illustration of the partitioning see Figure 1 on page 6 where marked vertices are drawn black and intermediate marked vertices gray. Hence, the number of removed edges is at most

$$\begin{aligned} & 2\lfloor \log(|V|) \rfloor + 2\left\lfloor \log\left(\left\lfloor \frac{|V|}{2} \right\rfloor\right)\right\rfloor + 2\left\lfloor \log\left(\left\lceil \frac{|V|}{2} \right\rceil\right)\right\rfloor \\ & \qquad \leq \; 2\lfloor \log(|V|) \rfloor + 2\lfloor \log(|V|) \rfloor - 2 + 2\lfloor \log(|V|) \rfloor \\ & \qquad \leq \; 6\lfloor \log(|V|) \rfloor - 2. \end{aligned}$$

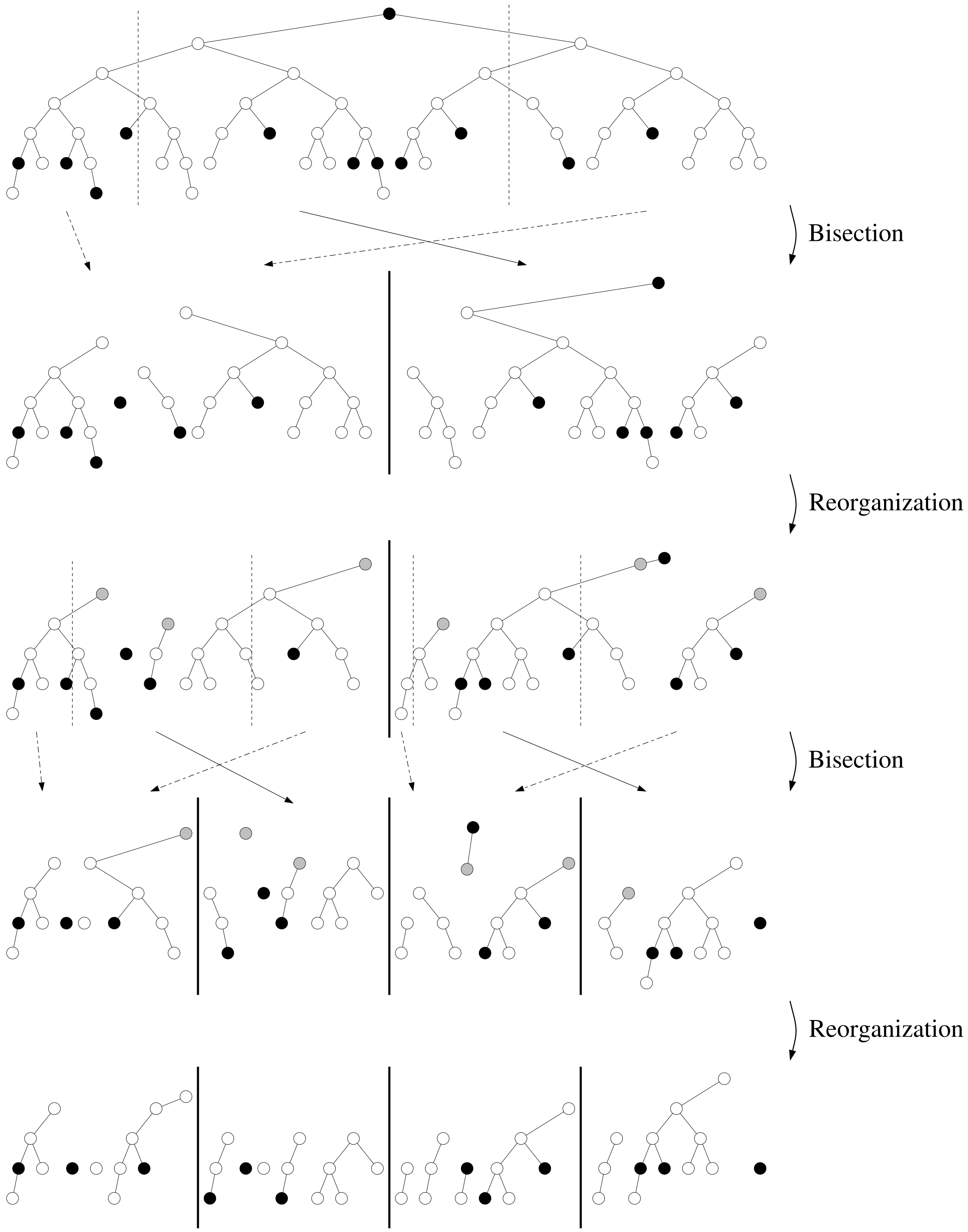

Figure 1: Partitioning a Forest

In the first bisection we remove at most $2\lfloor\log(|V|)\rfloor$ edges, and hence each of the newly created forests contains at most $2\lfloor\log(|V|)\rfloor$ partition vertices. The number of marked vertices in each of the newly created forests is at most $\lceil\frac{|V'|}{2}\rceil$. Since we mark all unmarked partition vertices of the first bisection, the number of marked vertices and

1 Preprocessing
  1.1 Reorganize the tree
  1.2 Compute the inorder numbering of the reorganized tree
2 Main Algorithm
  2.1 Fill up the root of the weighted quadtree using vertices with a small neighborhood.
  2.2 Loop:
    2.2.1 Assign partition and marked vertices to the corresponding weighted quadtree node
    2.2.2 Fill up the node with additional vertices
    2.2.3 Compute for each mapped vertex its position in the hypercube
    2.2.4 Mark the unmapped neighbors of mapped vertices
    2.2.5 Decompose the forest into four 'balanced' parts
    2.2.6 Reorganize the new forests and compute the new inorder numbering
  2.3 Route each vertex to its computed position in the hypercube
  2.4 Determine, for each vertex, the position in the hypercube of its neighbors

Figure 2: The Embedding Algorithm

partition vertices in each forest after the partitioning is at most

$$\left\lceil \frac{\lceil \frac{|V'|}{2} \rceil + 2\lfloor \log(|V|) \rfloor}{2} \right\rceil + 2\left\lfloor \log\left(\left\lceil \frac{|V|}{2} \right\rceil\right)\right\rfloor \leq \left\lceil \frac{|V'|}{4} \right\rceil + 3\lfloor \log(|V|) \rfloor. \tag{1}$$

■

3.2 The Embedding

The algorithm for the embedding proceeds in approximately $\frac{1}{2}\log(n)$ stages. Each stage is associated with a level of the weighted quadtree. At the first stage, we fill the root of the weighted quadtree with vertices and mark all unmapped neighbors of the mapped vertices. Then, we decompose the tree into four parts using Lemma 3.2 and associate the four parts with the four children of the root. The sizes of the four parts differ by at most 1.

In the following stages, we map the partition vertices and the marked vertices to the corresponding child node in the weighted quadtree. Then, we fill up each child node with arbitrary vertices from the binary tree in such a way that each mapped vertex has at most two unmapped neighbors. Again, we mark all unmapped neighbors of mapped vertices and decompose the forest into four 'balanced' forests as above.

To ensure dilation 8, we have to show that the number of vertices mapped to a node of the weighted quadtree does not exceed its capacity, or that such a conflict can be resolved.

In order to compute the number of tree vertices mapped to a single node in the weighted quadtree let $n(l)$ be the maximum number of marked vertices and partition vertices mapped to a single node of the weighted quadtree at level l, and let $f(l)$ be the size of a forest partitioned at a node of the weighted quadtree at level l. An obvious upper bound for $f(l)$ is $\lfloor \frac{2^d - c(1)}{4^{l-1}} \rfloor$. The number of marked vertices in a forest corresponding to a node at level l before the partitioning is at most $2c(l)$, since each mapped vertex has at most two unmapped neighbors. For our purpose, we choose $d = \lceil \log(n) \rceil$ and $k = 2$. Now we can establish the following inequalities using Lemma 3.2.

- If d is odd, we use a full weighted quadtree of height $h{=}\frac{d-3}{2}$, and we obtain for $l{\geq}3$:

$$\begin{aligned}
n(l) &\leq 3\left\lfloor \log\left(f(l-1)\right)\right\rfloor + \left\lceil \frac{1}{4}\left(2c(l-1)\right)\right\rceil \\
&\leq 3\left\lfloor \log\left(\frac{2^d-1}{4^{l-2}}\right)\right\rfloor + \frac{1}{2}\left(8\left(3(h-(l-1)+1)-1\right)\right) \qquad (2)\\
&= 3(d-2l+3)+12(h-l+1)+8 \\
&= 3(2h-2l+6)+12(h-l+1)+8 \\
&= 18(h-l+1)+20.
\end{aligned}$$

  Since the capacity of a node on level l of a full weighted quadtree is $24(h{-}l{+}1){-}8$, $n(l)$ should be less than this expression. This is the case for $l{\leq}h{-}4$.

- If d is even, we use a partial weighted quadtree of height $h{=}\frac{d-2}{2}$, and we obtain for $l{\geq}3$:

$$\begin{aligned}
n(l) &\leq 3\left\lfloor \log\left(f(l-1)\right)\right\rfloor + \left\lceil \frac{1}{4}\left(2c(l-1)\right)\right\rceil \\
&\leq 3\left\lfloor \log\left(\frac{2^d-1}{4^{l-2}}\right)\right\rfloor + \frac{1}{2}\left(8\left(3(h-(l-1)+1)-\frac{5}{2}\right)\right) \qquad (3)\\
&= 3\left(d-2l+3\right)+12\left(h-l+1\right)+2 \\
&= 3(2h-2l+5)+12(h-l+1)+2 \\
&= 18(h-l+1)+11.
\end{aligned}$$

  Since the capacity of a node on level l of a partial weighted quadtree is $24(h{-}l{+}1){-}$ -20, $n(l)$ should be less than this expression. This is the case for $l{\leq}h{-}5$.

We postpone bounding $n(2)$ (and $n(1)$) until the end of this subsection and first deal with the observation that in both cases we map more vertices to a node at the highest numbered four levels than the capacities of these nodes allow.

The excess is shifted downward one level by distributing the excess of a node to its four children. As we will show below, each vertex is shifted down at most one level. We mention here that edges between unshifted vertices already have dilation at most 8, as follows from Lemma 2.1.

Let a tree vertex that needs to be shifted downward initially be mapped to the node in the weighted quadtree represented by α. It is remapped to one of its children represented by $\alpha\alpha'$, $\alpha'{\in}\{0,1\}^2$. For the shifted vertices we only use the hypercube locations in $S'_{\alpha\alpha'}$.

To see that this suffices to achieve dilation $\leq$9, let v and w be adjacent vertices. The possible hypercube locations for a pair of adjacent tree vertices are illustrated in Figure 3, where each $*$ can be either 0 or 1. Note that hypercube locations of a form given by row (iii) or (vi) are only used by vertices shifted downward.

Case 1: We assume that initially w is mapped to a child of the node receiving v. Hence vertex v is finally mapped to a hypercube location whose label has a form given by row (i), (ii), or (iii), and w is finally mapped to a hypercube location whose label has a form given by the row (iv), (v), or (vi) of Figure 3. Since the strings in segment A agree, it is easy to see that we obtain dilation at most 9.

	A	B					C	D	
(i)	α	1	$0\longleftrightarrow 010 \longleftrightarrow 0$					$*$	$*$
(ii)	α	$*$	1	$0\longleftrightarrow 010 \longleftrightarrow 0$				$*$	$*$
(iii)	α	$*$	$*$	1	$0\longleftrightarrow 010 \longleftrightarrow 0$			$*$	$*$
(iv)	α	$*$	$*$	1	$0\longleftrightarrow 010 \longleftrightarrow 0$			$*$	$*$
(v)	α	$*$	$*$	$*$	1	$0\longleftrightarrow 010 \longleftrightarrow 0$		$*$	$*$
(vi)	α	$*$	$*$	$*$	$*$	1	$0\longleftrightarrow 010 \longleftrightarrow 0$	$*$	$*$

Figure 3: Possible Hypercube Locations of Two Adjacent Tree Vertices

Case 2: Suppose vertices v and w are initially mapped to the same node or to siblings of a weighted quadtree. The vertices v and w are then finally mapped to hypercube locations whose label has a form given by row (iv), (v), or (vi). Again, it is easy to see that this results in dilation at most 9.

We now prove that the sets $S'_{\alpha\alpha'}$ are sufficiently large, i.e., the remapping can be done in a 1-1 manner. This can be seen from the tables in Figure 4. Here, $h(l)=h-l+1$ is the height of a node at level l, $\Delta(l)=\max\{0, n(l)-c(l)+s(l)\}$ is the maximal excess of a node at level l, $s(l)=\lceil\frac{\Delta(l-1)}{4}\rceil$ is the maximal number of vertices shifted downward from a node at level $l-1$ to a node on level l, and $c'(l)=|S'_\alpha|$ is the capacity of a node at level l represented by α to take up vertices being shifted downward. Since the sum of the capacities of all nodes in a subtree rooted at a node at level l of the weighted quadtree is another upper bound for $f(l)$, we obtain $f(l)\leq\sum_{i=l}^{h}4^{i-l}c(i)$. Using this bound for $f(l)$ we obtain an upper bound for $n(l)$ as shown in the tables of Figure 4. Recall that $n(l)\leq 3\lfloor\log(f(l-1))\rfloor+\frac{1}{2}c(l-1)$. The values in parentheses represent improvements over the values given by the above formulae. These improvements stem from the fact that $n(h)\leq f(h)\leq c(h)$. Since obviously $s(l)\leq c'(l)$, we obtain an embedding with dilation at most 9.

To obtain dilation 8, we have to select the vertices being shifted down more carefully. Without loss of generality, we assume that each partition vertex is unmarked, since if a partition vertex is marked we have counted this vertex at least twice in our estimate for $n(l)$. We will shift down only unmarked partition vertices which are produced in one of the last two bisections in the proof of Lemma 3.2.

As indicated by the tables in Figure 4 there are enough such unmarked partition vertices. Here, $p(l)$ is the maximal number of partition vertices produced in one of the last two bisections of the partitioning mapped to a node at level l. From the estimate of the number of partition vertices (cf. inequality (1)) in the proof of Lemma 3.2 we

$h(l)$	$c(l)$	$n(l)$	$s(l)$	$\Delta(l)$	$c'(l)$	$p(l)$
5	112	104	0	0	40	24
4	88	86	0	0	32	20
3	64	68	0	4	24	16
2	40	50	1	11	16	12
1	16	(16)	3		8	8

full weighted quadtree

$h(l)$	$c(l)$	$n(l)$	$s(l)$	$\Delta(l)$	$c'(l)$	$p(l)$
5	100	95	0	0	36	22
4	76	77	0	1	28	18
3	52	59	1	8	20	14
2	28	41	2	15	12	10
1	4	(4)	4		4	4

partial weighted quadtree

Figure 4: Weighted Quadtree Statistics

obtain $p(l){=}2\lfloor\log(f(l))\rfloor$. As can be seen from the tables in Figure 4, $\Delta(l){\le}p(l)$ except for $h(l){=}2$ in the case of a partial weighted quadtree. However, it can easily be verified that adjacent vertices initially mapped to nodes at one of the three bottom levels of a partial weighted quadtree are finally mapped to hypercube locations with a distance of at most 8.

We note that only edges between unmarked partition vertices and their neighbors across partition edges can have dilation 9. All other neighbors of unmarked partition vertices are initially mapped to the same node, and corresponding edges have a dilation of at most 7.

On the other hand, a partition edge has dilation at most 8 if one of the following conditions holds: the neighbor across the partition edge is also being shifted down, or the neighbor is not mapped to the node represented by $\alpha\bar{\beta}$ assuming that $\alpha\beta$ represents the node to which the partition vertex itself is mapped, where $\alpha{\in}(\{0,1\}^2)^*$, $\beta{\in}\{0,1\}^2$, and $\bar{x}$ is the bit complement of x. Hence, a partition edge can require dilation 9 if and only if its incident partition vertices are initially mapped to weighted quadtree nodes represented by $\alpha\beta$ and $\alpha\bar{\beta}$, and exactly one of these vertices has been shifted. We call such a pair of nodes *complementary nodes*.

Recall that adjacent partition vertices are mapped to hypercube locations with labels as given in rows (iv) - (vi) of Figure 3. Since we only shift partition vertices produced in one of the last two bisection of the partition, the labels of their hypercube locations agree in the first bit of segment B. Thus, such a pair of adjacent partition vertices cannot be mapped to a pair of complementary nodes, and we achieve for our embedding a dilation of at most 8.

We also need a bound for the maximal value of $n(2)$. In order to minimize $n(2)$ we map vertices with a small neighborhood to the root of the weighted quadtree. We describe a method proceeding in steps until the capacity of the root is exhausted. If the grandparent of a deepest leaf or its sibling has at least three descendants then take the grandparent or its sibling, respectively, and all its descendants, otherwise take the great-grandparent of this deepest leaf and all its descendants. In either case, at most seven and at least four vertices are taken from the tree and at most one neighbor remains, except in the last step. Here, we take at most six leaves which have at most one neighbor each.

To analyze the ratio of the number of neighbors of removed tree vertices to the number of removed tree vertices, let x denote the number of vertices taken in any but the last step, and let y denote the number of vertices taken in the last step. Further, let N_x and N_y denote the number of neighbors remaining in the tree in the corresponding step. We obtain the following bound:

$$\frac{N_x + N_y}{x+y} \le \frac{\frac{1}{4}x + y}{x+y} \le \frac{1}{4} + \frac{3y}{4(x+y)} \le \frac{1}{4} + \frac{9}{2(x+y)}.$$

An easy computation shows that for all trees with more than 256 vertices (i.e. $d{\ge}9$) this ratio is at most $\frac{1}{3}$, since the capacity of the root ($=x{+}y$) is at least 96. With this observation we obtain the following estimate for $n(2)$ in case of a full weighted quadtree:

$$n(2) \le 6(h+1) + \frac{1}{4}\cdot\frac{1}{3}\cdot 32h = \frac{26}{3}\cdot h + 6.$$

It follows that $n(2)$ is less than or equal to the capacity of a node at level 2 of a full weighted quadtree if $h \geq 3$, i.e. $d \geq 9$. For a partial weighted quadtree we obtain the following estimate:

$$n(2) \leq 6\left(h + \frac{1}{2}\right) + \frac{1}{4} \cdot \frac{1}{3} \cdot 8(4h-2) = \frac{26}{3} \cdot h + \frac{13}{3}.$$

Again, $n(2)$ is less than or equal the capacity of a node at level 2 of a partial weighted quadtree if $h \geq 4$, i.e. $d \geq 10$.

Of course, trees of size less than 257, i.e. $d \leq 8$, have a trivial embedding of dilation 8 into their optimal hypercubes.

3.3 Congestion of the Embedding

We restrict our attention to achieving constant node-congestion, since the edge-congestion is less than or equal to the node-congestion.

Consider two adjacent vertices of the binary tree which are mapped to the hypercube locations labeled v and w. We decompose the label into four segments A, B, C, and D. Segment D consists of the last two bits. Segment C is the longest suffix before segment D such that it contains at most one 1 in both v and w. The remainder splits into segments A and B such that segment A is the longest common prefix, where v and w agree. Recall that the hypercube locations of adjacent tree vertices can differ only in segments B and D, and in at most 2 positions of segment C (cf. Figure 5 for the case $v, w \in S$). Also note that segments A and B can be empty.

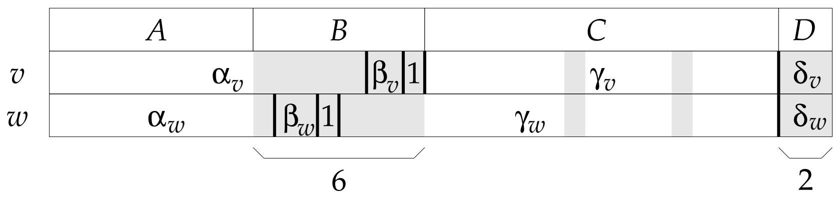

Figure 5: Hypercube Locations of Two Adjacent Tree Vertices

For a path $p_{v,w}$ from v to w, we call v the *lower* endpoint of $p_{v,w}$, if $v = \alpha_v \beta_v 1 \gamma_v \delta_v \in S$, $w = \alpha_w \beta_w 1 \gamma_w \delta_w \in S$ (cf. Figure 6) and $|\gamma_v| < |\gamma_w|$, or if $v \in S$ and $w \in T$. Otherwise, if $v, w \in S$ and $|\gamma_v| = |\gamma_w|$ or if $v, w \in T$ we arbitrarily select one endpoint of the path $p_{v,w}$ to be the lower endpoint. To construct a shortest path $p_{v,w}$ from v to w in the hypercube, we proceed in two phases. Without loss of generality, we assume that v is the lower endpoint. In the first phase, we flip those bit positions in segments B and D that have to be changed. First, we flip 0's to 1's and then 1's to 0's. In the second phase, we first flip the bit position in segment C which has to be changed from 0 to 1, if it exists. Then, we flip the bit position in segment C that has to be changed from 1 to 0.

To obtain an upper bound for the node-congestion, we consider a fixed hypercube location u. We count how often u can be an inner vertex of an image of a tree edge. If this number is bounded by c, the node-congestion is bounded by $c+3$, since the degree of a tree vertex is at most 3.

First, we consider paths that hit u and whose both endpoints belongs to S. Note that such a path can hit u only if its label contains at least two 1's in the first $d-2$ positions. While flipping bits in segments B and D, u and the label of the lower endpoint agree

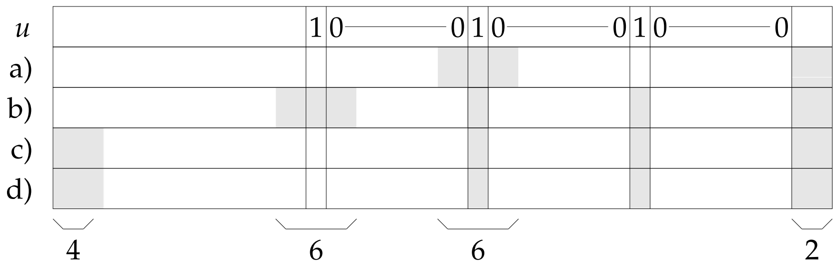

Figure 6: Possible Lower Endpoints of a Path Hitting u

except possibly in the shaded segments in row a) of Figure 6. Since the length of the shaded segments is bounded by a constant, at most a constant number of paths can hit u during the first phase.

For the second phase, we assume that there exists a bit position in segment C which has to change from 0 to 1. Otherwise, we are done. After flipping this bit position, u and the label of the lower endpoint agree except possibly in the shaded segments in row b) of Figure 6. Again, since the length of the shaded segments is bounded by a constant, at most a constant number of paths can hit u during the second phase.

Finally, we consider paths hitting u where at least one endpoint belongs to T. Recall that in this case segment B is contained in the first 4 positions of a label or is empty. Hence, u and the label of the lower endpoint agree except possibly for the shaded segments in row c) of Figure 6 while flipping bits in segments B and D. After flipping 0 to 1 in segment C, u and the label of the lower endpoint agree except possibly for the shaded segments in row d) of Figure 6. Thus, there again exists at most a constant number of paths hitting u.

Altogether, we have shown that for any hypercube location u at most a constant number of paths can hit u, finishing the proof.

4 An Implementation on the Hypercube

In this section, we show that the algorithm of the previous section can be implemented on a hypercube with an asymptotic time complexity bounded by the larger of $\log^2(n)$ and the time required by list ranking. The description of our hypercube algorithm consists of three parts: the preprocessing, the main algorithm, and a procedure for decomposing forests.

4.1 Preprocessing

The preprocessing algorithm is as given in Figure 7. To analyze its complexity, we use the following notations. Let $\mathrm{T}_{\mathsf{Sort}}(n)$ denote the time required to sort n items on a hypercube of dimension $\lceil\log(n)\rceil$ stored one element per processor. As shown in [CP90] $\mathrm{T}_{\mathsf{Sort}}(n){=}O(\log(n)\,\mathrm{loglog}^2(n))$.

Also, let $\mathrm{T}_{\mathsf{LR}}(n)$ denote the time for solving the list ranking problem for a list of length n on a hypercube of dimension $\lceil\log(n)\rceil$. A simple simulation of a PRAM algorithm [AM88, CV86a, CV86b] implies $\mathrm{T}_{\mathsf{LR}}(n){=}O(\log^2(n)\,\mathrm{loglog}^2(n))$. By a more

1 For each vertex, compute the size of its subtree, its depth, and its inorder number; also compute the height of the tree
2 For each vertex, compute the following information about its neighborhood in the tree:
  2.1 the inorder numbers for its children and its parent
  2.2 the size of the subtrees of its children and grandchildren
3 Mark and remove vertices to fill up the root
4 Reorganize the tree
5 Compute the new inorder numbering of the reorganized tree
6 Route each vertex to the location in the hypercube given by its inorder number
7 Repeat step 2.1 for the reorganized tree

Figure 7: Preprocessing Stage of the Hypercube Algorithm

sophisticated algorithm (as shown in Appendix A) we obtain

$$\mathrm{T}_{\mathsf{LR}}(n){=}O(\log^2(n)\,\mathrm{logloglog}(n)\log^*(n)).$$

For the preprocessing step, we use the Euler contour path technique. For each vertex v of a binary tree, we introduce three vertices LEFT(v), RIGHT(v), and LOWER(v) (called list vertices). We define recursively how to link together the list vertices. Consider the subtree induced by some vertex v. First, if v has a left child, link LEFT(v) with the beginning of the Euler contour path of its left subtree and the end of this Euler contour path with the list vertex LOWER(v), and link LEFT(v) with LOWER(v) otherwise. Next, if v has a right child, link LOWER(v) with the beginning of the Euler contour path of its right subtree and link the end of this Euler contour path with RIGHT(v), and link LOWER(v) with RIGHT(v) otherwise. Clearly, using list ranking and sorting, a parallel prefix operation on the Euler contour path can be performed in time $O(\max\{\mathrm{T}_{\mathsf{Sort}}(n), \mathrm{T}_{\mathsf{LR}}(n)\})$.

To compute, for each vertex, the size of its subtree in step 1 of the preprocessing, we mark LEFT(v) with 1 and v's other list vertices with 0. By computing the parallel prefix sum along the Euler contour path, we obtain, for each tree vertex v, the size of its subtree by subtracting the value computed at LEFT(v) from the value computed at RIGHT(v), and adding 1.

To compute the depth of each vertex, we mark LEFT(v) with 1, LOWER(v) with 0, and RIGHT(v) with -1 for all vertices v. The depth of a vertex v is the value LEFT(v) as determined by parallel prefix summation. The maximum depth of a vertex also determines the height of the tree.

Finally, if, for all vertices v, we mark LOWER(v) with 1 and the other list vertices with 0, a parallel prefix sum operation produces the inorder numbering of the tree.

Step 2 of the preprocessing consists of a constant number of routing problems which can all be solved using sorting.

Step 3 can be solved in time $O(\log^2(n))$ by finding $c(1){=}O(\log(n))$ vertices with a small neighborhood. Each group of at most seven vertices can be detected in time $O(\log(n))$ using parallel prefix operations. This can be seen as follows. Let h' be the height of the tree. We mark each vertex in the tree at depth $h'{-}2$ if it has more than 3 descendants . If in this step no vertex gets marked, we mark all vertices at depth $h'{-}3$ whose subtree has a size of at most 7. As mentioned earlier, at least one vertex will get marked. Next, we remove one marked vertex and its subtree from the tree. The height of the resulting tree can easily be recomputed using parallel prefix operations, as can the subtree sizes, using logarithmic time.

1 Fill the root up to capacity with the vertices selected in the preprocessing step and compute its hypercube location

2 Loop:

  2.1 Map the partition and marked vertices to the corresponding weighted quadtree nodes
  2.2 Fill the weighted quadtree nodes up to capacity using additional vertices
  2.3 For each mapped vertex, compute its location in the hypercube
  2.4 Mark the neighbors of mapped vertices
  2.5 Decompose each forest into four parts such that each part contains one quarter of marked vertices as well as one quarter of all vertices, using the procedure Decompose-Forest
  2.6 Divide the cube into four subcubes and route each newly created forest to its own subcube

3 Route the tree vertices to their computed locations in the hypercube and determine, for each vertex, the locations of its neighbors

---

Figure 8: Main Part of Hypercube Algorithm

For step 4 of the preprocessing we note that whether a vertex is a left or a right vertex in the tree only depends on how often the path from the root to the vertex branches to a child with a smaller subtree than its sibling. If this number is even (resp., odd) the vertex is a left (resp., right) vertex. Thus, we mark LEFT(v) with 1, RIGHT(v) with -1, and LOWER(v) with 0 if the size of v's subtree is less than that of its sibling, and we mark these list vertices with 0 otherwise. Again, using a parallel prefix sum, we find how often the path from the root to a vertex branches to the smaller subtree.

As described above, step 5 can also be implemented using parallel prefix operations on the Euler contour path. Steps 6 and 7 of the preprocessing consist of a constant number of routing problems which can be solved using sorting.

Altogether, we can implement the preprocessing in time $O(\max\{\mathrm{T}_{\mathsf{LR}}(n), \mathrm{T}_{\mathsf{Sort}}(n)\})$, where n is the size of the tree to be embedded.

4.2 The Main Algorithm

The main algorithm is as stated in Figure 8. The first step can easily be implemented in logarithmic time using parallel prefix operations, since we only have to compute the hypercube locations of the vertices marked in the preprocessing step. Step 3 can obviously be solved in time $O(\mathrm{T}_{\mathsf{Sort}}(n))$.

Each iteration of the loop in step 2 performs the assignment of tree vertices to the nodes at the corresponding level of the weighted quadtree. Thus, the i-th iteration is performed in 4^{i-1} distinct $(d-2i+2)$-dimensional subcubes, where each subcube contains a forest of size at most $\lceil\frac{n}{4^{i-1}}\rceil$. For the following discussion, we consider a fixed iteration of the loop and we denote by m the size of the forest contained in some subcube.

Step 2.1 can be solved in time logarithmic in m using parallel prefix operations. For step 2.2, in order to fill up the weighted quadtree node, we need a set of tree vertices such that each vertex has at most two neighbors. We mark all vertices having degree 1 or 2. Using a parallel prefix operation, we can easily select the appropriate number of these vertices using time logarithmic in m. Since at most half the vertices in a forest of binary trees can have degree 3, there are sufficiently many vertices with degree at most 2.

Step 2.3 can be computed in time $O(\log(m))$ using a parallel prefix operation. In step 2.4, we have to solve a routing problem since, for each mapped vertex, we have

1 Decompose the forest into two forests, such that each newly created forest contains one half of the marked vertices as well as one half of all vertices
2 Update the information about the size of subtrees of the children and grandchildren:
  2.1 Compute and broadcast the size of the removed subtree(s) in each tree
  2.2 Update the information about the size of subtrees of its children and grandchildren
3 Reorganize the newly created forests:
  3.1 For each vertex, compute the number of vertices on the path from the root to the vertex, for which the left subtree became smaller than the right subtree (this number is called the index of the vertex)
  3.2 Reorganize each tree in the forest by concentrating the vertices with odd index in reverse order and concentrate the even indexed vertices at the end
4 Update the inorder numbering:
  4.1 For each vertex, compute its new inorder number
  4.2 For each vertex, update the information about its children and its parent
5 Decompose each of the two newly created forests in their respective subcube using step 1 through step 4

Figure 9: Procedure Decompose-Forest

to mark its (unmapped) neighbors. As the number of neighbors of mapped vertices is $O(\log(m))$, this routing problem can be solved sorting a small set (using, e.g., sparse enumeration sort [NS82]), requiring only time logarithmic in m.

In the next subsection, we will show that the procedure Decompose-Forest can also be implemented in time logarithmic in m. Using monotone routing, step 2.6 requires time $O(\log(m))$.

Since there are $O(\log(n))$ stages altogether the time complexity of the main part of the algorithm is bounded by $O(\log^2(n))$.

4.3 Decomposing a Forest

In this subsection, we describe in detail how to decompose a forest in step 2.5 of the main algorithm. The procedure Decompose-Forest is listed in Figure 9. For our algorithm, we use routines for *reverse concentration routing*, which concentrates marked elements in an interval of hypercube locations in the same way as concentration routing but in reverse order. On a d-dimensional hypercube this can be done in time $O(d)$.

The inorder string can be bisected in constant time, if we preprocess it in the following manner. Using parallel prefix computation, we compute the number of marked vertices in each prefix of the string. After splitting it in the middle, we shift the second half in such a way that corresponding locations in the two halves are adjacent in the hypercube. Hence, the number of marks in the sliding window used in the proof of Theorem 3.1 can be computed using constant time. We can now determine the correct position of the window and broadcast its endpoints in time logarithmic in m. Thus, step 1 requires time $O(\log(n))$.

For simplicity, we now consider only one tree of the forest after the bisection, and we note that all computations are executed in parallel for all such trees. The operations required are parallel prefix computation, broadcasting, concentration routing, and reverse concentration routing. Since the trees are stored in inorder, we can perform these operations using segmented parallel prefix computations and monotone routing. In the case of the reverse concentration routing we have to be more careful, since the reversal takes place only within each tree. We determine, for all trees in parallel, a largest subcube within the interval belonging to the tree, and use this subcube to perform the reversal. Since the size of this subcube is at least a quarter of the length of the interval, a constant number of phases will suffice.

We call a vertex *pruned* iff it lost at least one child by the bisection. Since a subtree is removed from a tree only at the right end or left end of its inorder string, there are at most two pruned vertices in each newly created tree. If we removed subtrees at both ends of the inorder string, we split the tree into the left and right subtree of the root. After reorganizing these trees independently, we then recombine them. Thus, we may assume that each tree contains at most one pruned vertex, and we call the path from the root to a pruned vertex the *trunk* of the tree. In the following we also assume without loss of generality that the pruned vertex is the leftmost vertex of its tree.

The next step is to recompute, for each vertex, the size of its subtree. Each vertex knows the size of its left and right subtree before partitioning and the (old) inorder numbers of the leftmost and rightmost vertex after partitioning. Hence, each vertex can easily determine whether it lies on the trunk and whether it is the pruned vertex. The pruned vertex broadcasts the size of its subtree that was removed. Subsequently, each vertex on the trunk subtracts this value from the size of its own subtree, that of its left child and that of its leftmost grandchild. It follows that steps 2.1 and 2.2 require logarithmic time.

Next, the tree is reorganized as described in Section 3.1. After the partitioning, the property that the left (resp., right) subtree of a left (resp., right) vertex is larger than the right (resp., left) subtree may no longer hold. Call the corresponding vertex *reversed*. Note that all reversed vertices belongs to the trunk. The next step is to reestablish the above subtree property. Let the *index* of a vertex v be the number of reversed vertices on the path from the root to v. It is easy to see that a vertex belonging to the trunk and all vertices in its right subtree have the same index.

First, we consider an arbitrary reversed vertex v in a tree T. If we swap the children of each vertex in v's subtree, the vertex v is not longer reversed and no new reversed vertex is introduced in T. This operation corresponds to reversing the inorder substring of v's subtree within the inorder string of T. Hence, if we reverse all inorder substrings of the subtrees rooted at a reversed vertex, the corresponding tree is reorganized.

Let v be a vertex belonging to the trunk of a tree T, and let s be the inorder string of v's right subtree. After the above operation the inorder substring vs is unchanged if v's index is even, and it is reversed if v's index is odd. As the pruned vertex is the leftmost vertex in T, the vertices with an odd index move to the front of the inorder string, since the subtrees containing these vertices move to the left of the path from the root to the pruned vertex. Altogether, we obtain the inorder string of the reorganized tree, if we concentrate all vertices with an odd index in reverse order at the beginning, and concentrate the vertices with an even index at the end of the inorder string. This permutation can be realized in logarithmic time using algorithms for concentration and reverse concentration routing. Again, it is clear that logarithmic time suffices for steps 3.1 and 3.2.

This part of the algorithm is illustrated in Figure 10, where the larger subtree of a vertex is marked by shading. Thus, the vertices numbered 1, 5, 8, and 9 and their descendants in the right subtrees have to swap their subtrees.

It should also be noted that vertices mapped in some iteration of loop still remain

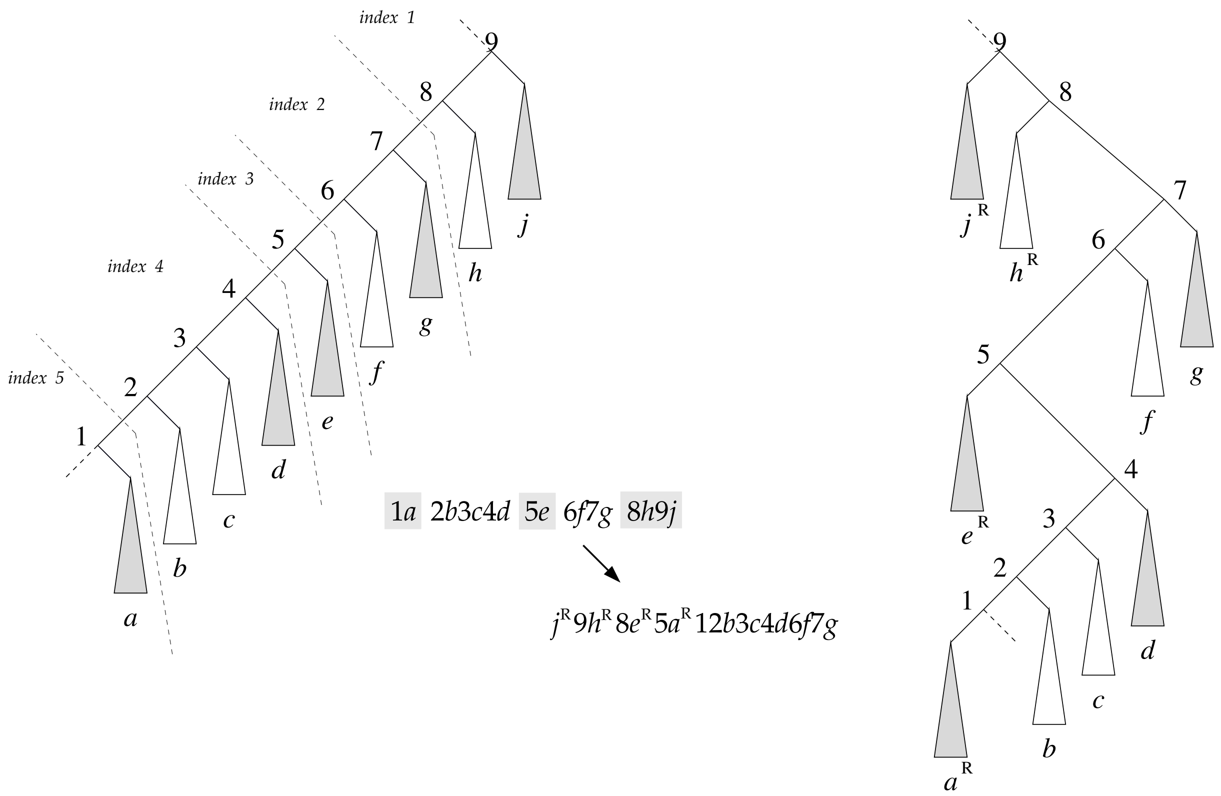

Figure 10: Reorganization of a Tree

in the forests in subsequent iterations. However, they wont be mapped again. Our estimation for $n(l)$ is still valid as can be seen from equation 2 and 3 on page 8.

To compute the new inorder number for each vertex, we only need a parallel prefix computation, because the tree vertices are now stored in inorder. Also, since each vertex knows the size of the subtrees of its children and grandchildren, it can easily compute the inorder number of its parent and its children. These operations again require at most logarithmic time.

Together we have proved the following theorem:

Theorem 4.1 *Let $T=(V,E)$ be an arbitrary binary tree of size n. There exists a one-to-one embedding of T into its optimal hypercube with dilation 8 and constant congestion. This embedding can be computed in time*

$$O\left(\max\left\{\mathrm{T}_{\mathsf{LR}}(n), \log^2(n)\right\}\right) = O\left(\log^2(n)\,\mathrm{logloglog}(n)\log^*(n)\right)$$

on the optimal hypercube. □

5 Conclusion

We have shown a simple algorithm to achieve a one-to-one embedding of arbitrary binary trees into their optimal hypercube with constant dilation and congestion. Our main result is that such embeddings can be computed on the hypercube itself in time $O(\max\{\log^2(n), \mathrm{T}_{\mathsf{LR}}(n)\})$.

Presently, the best known time bound for the list ranking problem on the hypercube is $\omega(\log^2(n))$, but we conjecture that this problem can actually be solved in time $O(\log^2(n))$. In this case, our embedding algorithm would also run in time $O(\log^2(n))$.

We can simulate computation on arbitrary binary trees in their optimal hypercubes suffering only a constant slowdown and the additional costs for the embedding as mentioned above. This simulation can be performed even if we have no information about the structure of the tree before running the algorithm.

Observe that any dynamic one-to-one embedding of a tree of size n into its optimal hypercube using a deterministic algorithm without migration leads to a dilation of $\Omega(\sqrt{\log(n)})$ in the worst case. This follows by a simple adaptation of a result in [LNRS89, LNRS92].

A List Ranking on the Hypercube

We consider the following list ranking problem: given a linked list of length n on a hypercube of dimension $\lceil \log(n) \rceil$ stored one element per processor, determine for each item in the list the number of its successors in the list. There are optimal EREW-PRAM algorithms using $\frac{n}{\log(n)}$ processors and $O(\log(n))$ time (see [AM88, CV86a, CV86b]).

[RJ89] presents a list ranking algorithm for a hypercube with p processors which runs in time $O(\frac{n}{p})$ when $n{=}\Omega(p^{1+\varepsilon})$, for any $\varepsilon{>}0$, and in time $O(\frac{n\log(n)}{p}+\log^3(p))$ otherwise. Since we need list ranking for lists with at most a constant number of items per processor the result of [RJ89] is suboptimal.

In the following we explain a faster hypercube algorithm for list ranking. It employs the well known path doubling strategy and obtains a better time bound of

$$\mathrm{T}_{\mathsf{LR}}(n){=}O(\log^2(n)\,\mathrm{logloglog}(n)\log^*(n)),$$

if there is at most one item per processor.

Let d denote the dimension of the hypercube. By $\mathrm{T}_{\mathsf{Sort}}(n,d)$ we denote the time required for sorting n items on the hypercube. From [CP90] and [NS82], we get the following time bounds:

$$\mathrm{T}_{\mathsf{Sort}}(n,d) = O\left(\min\left\{d\log^2(d), \frac{d\log(n)}{d-\log(n)+1}\right\}\right).$$

To select every second item of a linked list, we use the deterministic coin tossing technique of [CV86a, CV86b]. It proceeds in $O(\log^*(n))$ stages. Since, in each stage, communication occurs between neighbors in the linked list, we spend time $O(\mathrm{T}_{\mathsf{Sort}}(n,d)\log^*(n))$. The path doubling itself is also performed using sorting.

Hence we get the following time bound using the sorting algorithm in [CP90] for the initial phase and the algorithm from [NS82] for sorting few items:

$$\begin{aligned}
\mathrm{T}_{\mathsf{LR}}(n) &\leq \sum_{i=1}^{c} O\Big(\log\Big(\frac{n}{2^i}\Big)\,\mathrm{loglog}^2\Big(\frac{n}{2^i}\Big)\log^*\Big(\frac{n}{2^i}\Big)\Big) + \sum_{i=c+1}^{\lceil\log(n)\rceil} O\left(\frac{\log(n)\log\left(\frac{n}{2^i}\right)}{\log(2^i)}\log^*\Big(\frac{n}{2^i}\Big)\right)\\
&\leq O\left(\sum_{i=1}^{c}\log(n)\,\mathrm{loglog}^2(n)\log^*(n) + \log^2(n)\log^*(n)\sum_{i=c+1}^{\lceil\log(n)\rceil}\frac{1}{i}\right)
\end{aligned}$$

Picking $c = \frac{\log(n)}{\log\log^2(n)}$ yields:

$$\begin{aligned} \mathrm{T}_{\mathsf{LR}}(n) &\leq O\left(\log^2(n)\log^*(n) + \log^2(n)\log^*(n)\left(\log\log(n) - \log\left(\frac{\log(n)}{\log\log^2(n)}\right)\right)\right) \\ &\leq O\left(\log^2(n)\log\log\log(n)\log^*(n)\right) \end{aligned}$$

$\square$

References

[AW86] N. Alon, D. West: The Borsuk-Ulam Theorem and Bisection of Necklaces, *Proceedings of the American Mathematical Society*, Vol. 98 (1986), No. 4, 623–628.

[AM88] R. Anderson, G. Miller: Deterministic Parallel List Ranking, *VLSI Algorithms and Architectures, Proceedings of the 3rd Aegean Workshop on Computing 1988, LNCS 319*, 81–90.

[BCLR86] S. Bhatt, F. Chung, T. Leighton, A. Rosenberg: Optimal Simulations of Tree Machines, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science 1986*, 274–282.

[BCLR92] S. Bhatt, F. Chung, T. Leighton, A. Rosenberg: Efficient Embeddings of Trees in Hypercubes, *SIAM J. Comput.*, Vol. 21 (1992), No. 1, 151–162.

[BL84] S. Bhatt, T. Leighton: A Framework for Solving VLSI Graph Layout Problems, *J. Comput. System Sci.*, 28 (1984), 300–334.

[C90] F. Chung: Separator Theorems and Their Applications, in *B. Korte, L. Lovász, H. J. Prömel, A. Schrijver: Paths, Flows, and VLSI-Layout*, Springer-Verlag, 1990, 17–34.

[CV86a] R. Cole, U. Vishkin: Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, *Proceedings of the 18th ACM Symposium on Theory of Computing 1986*, 206–219.

[CV86b] R. Cole, U. Vishkin: Deterministic Coin Tossing with Applications to Optimal List Ranking, *Inf. Control* 70 (1986), 32–53.

[CP90] R. Cypher, G. Plaxton: Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing 1990*, 193–203.

[FM87] T. Feder, E. Mayr: An Efficient Algorithm for Embedding Complete Binary Trees in the Hypercube, 1987.

[GW85] C. Goldberg, D. West: Bisection of Circle Colorings, *SIAM J. Algebraic Discrete Math.*, 6 (1985), 93–106.

[L92] T. Leighton: Introduction to Parallel Algorithms and Architectures: Arrays – Trees – Hypercubes, Morgan Kaufmann Publishers, 1992.

[LNRS89] T. Leighton, M. Newman, A. Ranade, W. Schwabe: Dynamic Tree Embeddings in Butterflies and Hypercubes, *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Arcitectures*, 224–234.

[LNRS92] T. Leighton, M. Newman, A. Ranade, W. Schwabe: Dynamic Tree Embeddings in Butterflies and Hypercubes, *SIAM J. Comput.*, Vol. 21 (1992), No. 4, 639–654.

[MS88] B. Monien, H. Sudborough: Simulating Binary Trees on Hypercubes, *VLSI Algorithms and Architectures, Proceedings of the 3rd Aegean Workshop on Computing 1988, LNCS 319*, 170–180.

[NS82] D. Nassimi, S. Sahni: Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network, *Journal of the ACM*, 29 (3), 1982, 642–667.

[RJ89] K. W. Ryu, J. Jájá: List Ranking on the Hypercube, *Proceedings of the 1989 International Conference on Parallel Processing*, Vol. III, 20–23.

[SS85] Y. Saad, M. Schulz: Topological Properties of the Hypercube, Yale University Research Report RR-389, 1985.