Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

Volker Heun, Ernst W. Mayr

TUM-I9539
SFB-Bericht Nr.342/22/95 A
Dezember 1995

TUM–INFO–12-95-I39-350/1.–FI

Embedding Graphs with Bounded Treewidth into Optimal Hypercubes

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

Abstract

In this paper, we present a one-to-one embedding of a graph with bounded treewidth into its optimal hypercube. This is the first time that embeddings of graphs with a very irregular structure into hypercubes are investigated. The dilation of the presented embedding is bounded by $3\lceil\log((d+1)(t+1))\rceil+8$, where t denotes the treewidth of the graph and d denotes the maximal degree of a vertex in the graph. The given embedding is a generalization of our method to embed arbitrary binary trees into their optimal hypercubes given in [HM93]. Moreover, if the given graph has constant treewidth or is represented by a tree-decomposition of width t, this embedding can be efficiently implemented on the optimal hypercube itself. $\square$

1 Introduction

Hypercubes are a very popular model for parallel computation because of their regularity and their relatively small number of interprocessor connections. Another important property of an interconnection network is its ability to simulate efficiently the communication of parallel algorithms. Thus, it is desirable to find suitable embeddings of graphs representing the communication structure of parallel algorithms into hypercubes representing the interconnection network of a parallel computer.

The embedding of graphs with a regular structure, like rings, (multidimensional) grids, complete trees, binomial trees, pyramids, X-trees, meshes of trees and so on, have been investigated by many researcher, see, e.g., [Cha89, Cha91, CCCM93, HL73, FM87, SHL95, Sto86, Efe91]. Unfortunately, the communication structure of a parallel algorithm can often be very irregular. Embeddings of such irregular graphs have only been studied in the special case when the graph is an arbitrary binary tree.

For graphs whose structure is less regular, it is in general hard to decide whether there is a good embedding into a given host graph. Here, an embedding is considered as good if it has small dilation and load. Given a graph G and a positive integer d, it is $\mathcal{NP}$-complete to decide whether G is a subgraph of a d-dimensional hypercube, even if G is a tree [WC90]. Given a graph G and a positive integer d, it is also $\mathcal{NP}$-complete to decide whether G has a dilation 2 embedding into the d-dimensional hypercube [WC93]. The unboundedness of the degree of the given graph is essential in the proof. Hence, it might be possible to construct efficient embeddings if the degree of a graph is bounded.

For arbitrary binary trees, one-to-one embeddings into their optimal hypercubes with constant dilation and constant node-congestion have been constructed in [BCLR86, BCLR92]. The exact value for the dilation was not given, but it ranges between 12 and 15. Another one-to-one embedding of an arbitrary binary tree into its optimal hypercube with constant dilation is given in [MS88], but in this paper neither node-congestion nor edge-congestion is considered. In [MS88], the existence of embeddings with smaller dilation was claimed. However, the proof given in that paper is incomplete and it is still open whether the claimed dilation could be achieved. The embedding given in [HM93] yields dilation 8 and constant node-congestion. This is the best known bound on the dilation. Furthermore, this embedding can be efficiently computed on the hypercube itself. In [Hav84], Havel has conjectured that every binary tree has a one-to-one embedding into its optimal hypercube with dilation 2. This conjecture is still open. 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].

It has also been investigated whether balanced caterpillars of size 2^d are subgraphs of d-dimensional hypercubes. Caterpillars are binary trees whose vertices with degree 3 lie on a single path in the binary tree. This path is called the backbone of the caterpillar. All other paths growing from the backbone are called legs. Since a tree is bipartite graph, we call a tree balanced if both sets of the bipartition of the vertices have the same cardinality. In [HL86], it was shown that balanced caterpillars with legs of unit length are subgraphs of their optimal hypercubes. This was generalized in [BMUW93]; it was shown that balanced caterpillars with the additional property that the length of its legs have the same parity are subgraphs of their optimal hypercubes.

In this paper, we present a one-to-one embedding of graphs with bounded treewidth into their optimal hypercubes based on the tree-decomposition of the given graph. This is the first time that embeddings of graphs with an extremely irregular structure into hypercubes are investigated. The dilation of the presented embedding grows logarithmically in the treewidth of the graph and in the maximal degree of a vertex. Hence, for graphs with constant treewidth and constant degree, we exhibit a one-to-one embedding with constant dilation and node-congestion into their optimal hypercubes. Furthermore, if the graph is given by a tree-decomposition, we present an efficient construction of the embedding on the optimal hypercube. The embedding can be computed in time $O(\max\{\mathrm{T}_{\mathsf{LR}}(n), (d+t^2)\log^2(n)\})$, where $\mathrm{T}_{\mathsf{LR}}(n)$ denotes the time complexity for list ranking on the hypercube. If the given graph has constant treewidth, a minimal tree-decomposition for this graph can be computed on the optimal hypercube in time $O(\log^3(n)\,\mathrm{loglog}^2(n))$, using a result in [BH95].

On the other hand, we present some lower bounds which will give strong evidence that each embedding of a graph with treewidth t and maximal degree d into its optimal hypercube requires dilation $\Omega(\log(dt))$.

2 Preliminaries

2.1 Embeddings

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 the 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 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 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 $\frac{|V_H|}{|V_G|}$ is called the *expansion* of the embedding ϕ.

The *congestion* of an edge $e'{\in}E_H$ is the number of paths in $\{\phi_E(e) \mid e{\in}E_G\}$ that contain 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. Again, the *node-congestion* is the maximal congestion over all vertices in H.

2.2 Hypercubes and Trees

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. An edge belongs to dimension i, if the labels of the vertices incident to that edge differ in position i. 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 $\lceil\log(|V|)\rceil$[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 small node-congestion.

The *level* of a vertex v in a tree is the number of vertices on the path from the root to v. Hence, the level of the root is 1. The *height* of a tree T is the maximum level of a vertex

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

in T. A *complete* d-ary tree T of height h is a tree such that each internal vertex has exactly d children, and such that all leaves have the same level. A *nearly* complete d-ary tree T of size n is a subtree of the smallest complete d-ary tree of size greater equal n such that the following hold: T has exactly n vertices, all but one of the internal vertices have d children, and the levels of any two leaves differ by at most one.

2.3 Graphs with Bounded Treewidth

A *tree-decomposition* $\mathcal{D}_G$ of a graph $G{=}(V,E)$ is a pair (T,X) consisting of a tree $T{=}(S,F)$ and a set $X{=}\{X_s{\subseteq}V{:}s{\in}S\}$ such that the following three conditions are satisfied:

i) $\bigcup_{s\in S} X_s = V$,

ii) $\forall(v,w){\in}E : \exists s{\in}S : v,w{\in}X_s$,

iii) for each vertex $v{\in}V$ the induced subgraph of T on the set $\{s{:}v{\in}X_s\}$ is a tree.

T is also called a *decomposition tree* for G. The *width* of a tree-decomposition $\mathcal{D}_G{=}(T,X)$ is defined as $\max\{|X_s|{-}1{:}s{\in}S\}$. The *treewidth* of a graph $G{=}(V,E)$ is the minimum width over all tree-decompositions of G.

Given a graph G and an integer t, it is in general $\mathcal{NP}$-complete to decide whether G has treewidth at most t [ACP87]. On the other hand, for some special classes of graphs, e.g., permutation graphs, cotriangulated graphs, convex graphs, chordal bipartite graphs, and circle graphs there exist polynomial time algorithms to determine the treewidth (see, e.g., [Klo94]). For graphs with constant treewidth an algorithm for constructing a minimal tree-decomposition is described in [BH95]. This algorithm works in time $O(\log^2(n))$ using $O(n)$ operations on a EREW PRAM. Hence, given a graph G with constant treewidth t, a minimal tree-decomposition of width t for G can be computed on the hypercube in time $O(\log^3(n)\,\mathrm{loglog}^2(n))$.

The *size* of a tree-decomposition $\mathcal{D}_G{=}(T,X)$ is defined as the cardinality of X. Note that $|X|{=}|S|$, where $T{=}(S,E)$. Given a graph G with treewidth t, we always can find a small tree-decomposition for G.

Lemma 1 *Let $G{=}(V,E)$ be a graph with treewidth t. There exists a tree-decomposition $\mathcal{D}_G{=}(T,X)$ for G of width t such that $|X|{\le}|V|$.*

Proof:*(Sketch)* This follows from the fact that the class of graphs with treewidth k coincides with the class of partial k-trees (e.g., see [Klo94]). ■

A tree-decomposition $\mathcal{D}_G{=}(T,X)$ of a graph $G{=}(V,E)$ is called *binary* if T is a binary tree. It is possible to transform each tree-decomposition into a binary tree-decomposition as the next lemma shows. We will need the proof later to transform a tree-decomposition into a binary one.

Lemma 2 *Let $G{=}(V,E)$ be a graph with treewidth t. There exists a binary tree-decomposition $\mathcal{D}_G{=}(T,X)$ for G of width t and $|X|{\le}2|V|$.*

Proof: Consider a vertex s of the decomposition tree $T{=}(S,F)$ with $n{>}2$ children. Replace vertex s by a nearly complete binary tree with n leaves. For all internal vertices v of this nearly complete binary tree define $X_v{=}X_s$. Identify the n leaves of this nearly complete binary tree with the n children of s. Repeating this procedure until all vertices have at most two children yields a binary tree-decomposition for G. Since the number of internal vertices of a nearly complete binary tree is less than the number of its leaves, the size of the binary decomposition tree is at most twice the size of the given decomposition tree. ■

To establish some lower bounds on the dilation in the next section, we present some fundamental properties of graphs with bounded treewidth.

Lemma 3 *Let $G{=}(V,E)$ be graph containing a $t{+}1$-clique. Then the treewidth of G is at least t.*

Proof: Let $\mathcal{D}_G{=}(T,X)$ be a tree-decomposition for G and let $C{\subseteq}V$ be a $t{+}1$-clique in G. It is sufficient to show that there exist a vertex $s{\in}S$ in the decomposition tree $T{=}(S,F)$ such that $C{\subseteq}X_s$. This will be proved by induction on the clique size.

Induction base ($t{=}1$):
The claim follows immediately from the definition.

Induction step ($t{-}1{\to}t$):
Suppose the claim holds for cliques of size $t{\ge}2$. Let C be a clique of size $t{+}1$ and suppose for contradiction that the claim does not hold for C. Let $v{\in}C$ and define $C'{=}C{\setminus}\{v\}$. Let $S'{\subseteq}S$ be the set $\{s{\in}S{:}C'{\subseteq}X_s\}$. By induction, it follows that $S'{\ne}\emptyset$. Further, let $U{=}\{u{\in}S{:}v{\in}X_u\}$. Note that $S'{\cap}U{=}\emptyset$ and that the induced subgraphs on S' and U of T are subtrees of T. Consider the path $s_1,\ldots,s_l$ in T joining these subtrees such that $s_1{\in}S'$, $s_l{\in}U$ and $s_2,\ldots,s_{l-1}{\notin}S'{\cup}U$. Hence, there exists $w{\in}C{\setminus}X_{s_l}{\ne}\emptyset$. By condition *iii)* in the definition of a tree-decomposition, the edge $\{v,w\}$ has no representation in the decomposition tree T. Thus, we get a contradiction. ■

A graph $G{=}(V,E)$ has a s-*separator* (A,B,S), if there exists a partition of V into the sets A, B, and S such that the following conditions are satisfied:

i) $|S| = s$,

ii) $\frac{1}{3}(|V|-s) \le |A|, |B| \le \frac{2}{3}(|V|-s)$,

iii) S separates A from B, i.e., there are no edges between A and B.

It is well known that every graph of treewidth t has a $t{+}1$-separator (see, e.g., [Klo94]). To prove this fact, we first need a combinatorial Lemma:

Lemma 4 *Let $t_1 \leq \cdots \leq t_\ell \leq \frac{2n}{3}$ be real numbers such that $n = \sum_{i=1}^{\ell} t_i$. Then there exists an integer j such that $\frac{n}{3} \leq \sum_{i=1}^{j} t_i, \sum_{i=j+1}^{\ell} t_i \leq \frac{2n}{3}$.*

Proof: Choose j such that $\sum_{i=1}^{j} t_i \leq \frac{2n}{3} < \sum_{i=1}^{j+1} t_i$, implying that $\sum_{i=j+2}^{\ell} t_i < \frac{n}{3}$. It remains to show that $\sum_{i=1}^{j} \geq \frac{n}{3}$. If $l = j+1$ then the assumptions imply $t_{j+1} \leq \frac{2n}{3}$. Otherwise we obtain $t_{j+1} \leq \sum_{i=j+2}^{\ell} t_i \leq \frac{n}{3}$. In both cases we get $\sum_{i=j+1}^{\ell} t_i \leq \frac{2n}{3}$. Thus, $\sum_{i=1}^{j} t_i = n - \sum_{i=j+1}^{\ell} t_i \geq \frac{n}{3}$ finishing the proof. ∎

Let $G = (V, E)$ be a graph. For $W \subseteq V$, we denote by $G[W]$ the induced subgraph of G on W, i.e., $G[W] = (W, E \cap (W \times W))$.

Lemma 5 *Let $G = (V, E)$ be a graph with treewidth t, then G has a $t+1$-separator.*

Proof: From Lemma 4 it is sufficient to show that there exists a set $S \subseteq V$ of size $t+1$ such that each connected component of $G[V \setminus S]$ has size at most $\frac{n-(t+1)}{2}$. Let (T, X) be a tree-decomposition of width t for G, where $T = (W, F)$ is the decomposition tree. Without loss of generality we assume that for each vertex $w \in W$ $|X_w| = t+1$ and that for each edge $(v, w) \in F$ $|X_v \triangle X_w| = 2$.

Consider the following algorithm. Let $w \in W$ be a vertex of T. If the size of all connected components of $G[V \setminus X_w]$ are bounded by $\frac{1}{2}(n-t-1)$, choose $S = X_w$. Otherwise, let w' be a vertex adjacent to w such that $X_{w'}$ and the largest connected component have a common vertex. The algorithm continues with w' instead of w. By construction, the size of the largest connected components shrinks in each iteration of the algorithm. Hence, the algorithm terminates. ∎

2.4 Lower Bounds

First, we will show that every embedding of a graph with treewidth t and maximal degree $d \geq t$ into its optimal hypercube has dilation at least $\Omega(\log(d))$.

Lemma 6 *Let $T = (V, E)$ be a nearly complete d-ary tree of size n. Every one-to-one embedding of T into its optimal hypercube has dilation at least $\Omega(\log(d))$.*

Proof: Given a one-to-one embedding, consider two vertices $v_0, v_1 \in V$ which are mapped to the hypercube vertices whose labels differ in at least $\frac{1}{2}\log(n)$ positions. The existence of such a pair of hypercube vertices can be seen as follows. The number of vertices whose labels differ in at most $\lceil \frac{1}{2}\log(n) - 1 \rceil$ positions from a fixed vertex v_0 is at most

$$\sum_{i=0}^{\lceil \frac{1}{2}\log(n)-1 \rceil} \binom{\lceil \log(n) \rceil}{i} \leq 2^{\lceil \log(n) \rceil - 1} < n.$$

Since we consider a one-to-one embedding into the optimal hypercube, exactly n hypercube vertices are images of the mapping. Thus, there must exist a vertex v_1 mapped to a hypercube vertex whose label differs in at least $\lceil \frac{1}{2}\log(n) \rceil$ position from v_0. It is easy to see that the distance in T between any two vertices of T is at most $O(\frac{\log(n)}{\log(d)})$. Now we can conclude that at least one edge on the path from v_0 to v_1 has dilation at least $\Omega(\frac{\log(n)/2}{\log(n)/\log(d)}) = \Omega(\log(d))$. ■

Theorem 7 *Given integers n, d and t such that $t \leq d < n$. There exists a graph $T^n_{t,d}$ of size n, treewidth t and maximal degree d such that each one-to-one embedding of $T^n_{t,d}$ into its optimal hypercube requires dilation at least $\Omega(\log(d))$.*

Proof: We construct the graph $T^n_{t,d}$ from a nearly complete $d-1$-ary tree T_c of size n. The graph $T^n_{t,d} = (V, E)$ is a supergraph of T_c, where an edge $\{v, w\}$ between v and w is added if v and w are leaves of the same parent and if v and w are one of the t leftmost children of their parent. Lemma 3 implies that the treewidth of $T^n_{t,d}$ is at least t. On the other hand, it can easily be seen that the treewidth of $T^n_{t,d}$ is also at most t. Clearly, the size of $T^n_{t,d}$ is n, and the maximal degree of a vertex in $T^n_{t,d}$ is d. Since $T^n_{t,d}$ contains T_c as a subgraph, Lemma 6 implies that the dilation of each one-to-one embedding of $T^n_{t,d}$ into its optimal hypercube has dilation at least $\Omega(\log(d))$. ■

Given a graph $G=(V, E)$, we denote by $N(U) := \{v : (u, v) \in E \wedge u \in U\}$ the *neighborhood* of the set $U \subseteq V$. The *expansion* of a set of vertices $U \subseteq V$ is defined as $\frac{|N(U) \setminus U|}{|U|}$. An α-*expander* is a regular graph $G=(V, E)$ such that every set $U \subseteq V$ of size at most $\frac{|V|}{2}$ has expansion α, where α is a fixed constant independent of the size of the graph. Using probabilistic arguments, it easy to prove the existence of an infinite family of constant-degree α-expanders for some constant α (see, e.g., [Chu78, Pin73, Pip77]). Moreover, almost every regular random graph is an expander. On the other hand, it is quite hard to construct an infinite family of constant-degree α-expanders deterministically.

Following the proof in [BCLR92], we show a lower bound on the dilation for graphs with large treewidth.

Lemma 8 *Every one-to-one embedding of a constant-degree α-expander with n vertices into its optimal hypercube has dilation at least $\Omega(\alpha \log(n))$.*

Proof: Let $G=(V, E)$ be a constant-degree α-expander of size n. Consider a one-to-one embedding of G into its optimal hypercube of size N, implying that $N < 2n$. The removal of all edges of two fixed dimensions splits the hypercube into four subcubes. Also the embedded graph will be split into four parts. Each part of the split graph contains at most $\frac{N}{4} < \frac{n}{2}$ graph vertices. One part of the split graph has size at least $\frac{n}{4}$. Since G is an α-expander, at least $\alpha\frac{n}{4}$ graph edges traverse hypercube edges of the removed dimensions. Since there exists $\lfloor \frac{\log(N)}{2} \rfloor$ pairs of dimensions such that each dimension occurs at most once, we can conclude that altogether at least $\lfloor \alpha \frac{n \log(n)}{8} \rfloor$ hypercube edges are traversed by graph edges. Since the number of edges in G is $O(n)$, the dilation of the embedding is at least $\Omega(\alpha \log(n))$. ■

Lemma 9 *Every constant-degree α-expander of size n has treewidth $\Theta(n)$.*

Proof: Let $G{=}(V,E)$ be an α-expander of size n with degree d and treewidth t. By Lemma 5, consider a $t{+}1$-separator (A,B,S) for G. Without loss of generality, we assume that $\frac{n-(t+1)}{3}{\le}|A|{\le}\frac{n}{2}$. Since S separates A from B, we have $|N(A)|{\le}t$. On the other hand, G is an α-expander, which implies that $|N(A)|{\ge}\alpha{\cdot}|A|{\ge}\alpha\frac{n-t-1}{3}$. Altogether we get $t{\ge}\frac{\alpha(n-1)}{\alpha+3}{=}\Omega(n)$. Since each graph of size n has obviously treewidth $O(n)$, the claim follows. ■

Combining Lemma 8 and Lemma 9 yields the following theorem.

Theorem 10 *There exists an infinite family $\mathcal{G}$ of graphs with constant degree and treewidth $\Theta(n)$, where n is the size of the graph, such that every one-to-one embedding of a graph into its optimal hypercube has dilation at least $\Omega(\log(t))$.*

2.5 The (k,h,o)-tree

To construct our embedding, we use the data structure of a (k,h,o)-tree. The (k,h,o)-*tree* is a complete 2^k-ary tree of height h with integer node weights, also called the *capacities* of the nodes. We distinguish different types of a (k,h,o)-tree, which we call the (k,h,o)-tree of the o-th order. The only difference is in the capacities. For a (k,h,o)-tree of the o-th order the root has a capacity of $2^o(2^k(kh{-}k{+}1))$, and every other node at level l has capacity $2^o((2^k{-}1)(k(h{-}l{+}1){-}k{+}1){-}k)$. In the following, we call vertices of a (k,h,o)-tree *nodes*, and we denote the capacity of a node at level l by $c(l)$.

Lemma 11 *The total capacity of a (k,h,o)-tree is 2^{kh+o}.*

Proof: By the definition of the capacities we get:

$$\begin{aligned}
\sum_{l=1}^{h} c(l) &= 2^o(2^k(kh-k+1)) + \sum_{l=2}^{h} 2^{k(l-1)}\left[2^o((2^k-1)(k(h-l+1)-k+1)-k)\right] \\
&= \sum_{l=1}^{h} 2^{k(l-1)}\left[2^o((2^k-1)(k(h-l+1)-k+1)-k)\right] + 2^o(kh+1) \\
&= 2^o\left[(2^k-1)\left((kh-k+1)\frac{2^{kh}-1}{2^k-1} - k\sum_{l=0}^{h-1} l2^{kl}\right) - k\frac{2^{kh}-1}{2^k-1} + kh+1\right] \\
&= 2^o\Bigg[(kh-k+1)(2^{kh}-1) + kh + 1 \\
&\qquad -(2^k-1)k\frac{(h-1)2^{k(h+1)} - h2^{kh} + 2^k}{(2^k-1)^2} - k\frac{2^{kh}-1}{2^k-1}\Bigg]
\end{aligned}$$

$$\begin{aligned}
&= 2^o\Big[kh2^{kh} - kh - (k-1)2^{kh} + (k-1) + kh + 1 \\
&\qquad -k\frac{2^k h2^{kh} - h2^{kh} - 2^k 2^{kh} + 2^{kh} + 2^k - 1}{2^k - 1}\Big] \\
&= 2^o\Big[kh2^{kh} - (k-1)2^{kh} + k - kh2^{kh} + k2^{kh} - k\Big] \\
&= 2^{kh+o}
\end{aligned}$$

■

We now describe a mapping of a (k,h,o)-tree into its optimal hypercube such that each node of the (k,h,o)-tree occupies as many vertices of the hypercube as given by its capacity. Each node in a (k,h,o)-tree can be represented by a string in $(\{0,1\}^k)^*$ as follows. The empty string ε represents the root of the (k,h,o)-tree. If α represents a node v of the (k,h,o)-tree, then the strings $\alpha\beta$ for $\beta{\in}\{0,1\}^k$ represent the 2^k children of v from left to right. We define the following sets of hypercube locations, where α represents some arbitrary node in a (k,h,o)-tree:

$$\begin{aligned}
S_\alpha &:= \Big\{\alpha\beta\gamma\delta \in \{0,1\}^{kh+o} : \beta{\in}(0{+}1)^*1 \wedge |\beta|{\le}k \wedge \gamma{\in}0^*10^* \wedge \delta{\in}\{0,1\}^o\Big\} \\
T &:= \Big\{\gamma\delta \in \{0,1\}^{kh+o} : \gamma{\in}(0^*10^*{+}0^*) \wedge \delta{\in}\{0,1\}^o\Big\}
\end{aligned}$$

We also define the set $S{:=}\bigcup_\alpha S_\alpha$. The vertices of the binary tree mapped to the node of a (k,h,o)-tree 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. We will now show that the capacity of a node in the (k,h,o)-tree is equal to the cardinality of the set of vertices in the hypercube to which it is mapped.

Lemma 12 *For a (k,h,o)-tree we have*

i) $|T|{=}2^o(kh{+}1)$ *and*

ii) $|S_\alpha|{=}2^o((2^k{-}1)(k(h{-}l{+}1){-}k{+}1){-}k)$,

where α represents a node at level l. Furthermore, $|L_\alpha|{=}c(l)$.

Proof: Equality *i)* follows immediately from the definition of T. Since α represents a node of the (k,h,o)-tree at level l, we have $|\alpha|{=}(l{-}1)k$. By definition of S_α, we obtain $|\gamma|{=}kh{-}|\alpha\beta|{=}k(h{-}l{+}1){-}|\beta|$. Thus, we get:

$$\begin{aligned}
|S_\alpha| &= 2^o \sum_{\substack{\beta\in(0+1)^*1\\|\beta|\le k}} |\gamma| \\
&= 2^o \sum_{\substack{\beta\in(0+1)^*1\\|\beta|\le k}} (k(h-l+1) - |\beta|) \\
&= 2^o \sum_{i=0}^{k-1} 2^i[k(h-l+1) - (i+1)]
\end{aligned}$$

$$
\begin{aligned}
&= 2^o \left((2^k-1)(k(h-l+1)) - \frac{1}{2}\sum_{i=1}^{k} i2^i \right) \\
&= 2^o \left((2^k-1)(k(h-l+1)) - \frac{1}{2}\left(k2^{k+2}-(k+1)2^{k+1}+2\right) \right) \\
&= 2^o \left((2^k-1)(k(h-l+1)) - k2^k + 2^k - 1 \right) \\
&= 2^o \left((2^k-1)(k(h-l+1)-k+1) - k \right)
\end{aligned}
$$

The second claim follows immediately from the above equalities. ■

Furthermore, it can easily be verified that for every $\alpha{\neq}\alpha'{\in}(\{0,1\}^k)^*$ $S_\alpha{\cap}S_{\alpha'}{=}\emptyset$ and $S_\alpha{\cap}T{=}\emptyset$, and hence $L_\alpha{\cap}L_{\alpha'}{=}\emptyset$ for every $\alpha{\neq}\alpha'{\in}(\{0,1\}^k)^*$. Hence, for each string $s{\in}S{\cup}T$ there is a unique decomposition $s{=}\alpha\beta\gamma\delta$ as used in the definition of S_α and T.

Lemma 13 *Let v and w be two nodes in a (k,h,o)-tree such that their lowest common ancestor is at distance $\leq d$ from both v and w. The labels of any pair of corresponding vertices in the hypercube to which v and w are mapped differ in at most $k(d{+}1){+}o{+}2$ positions.*

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'	γ'	δ'
w:	α	α''	β''	γ''	δ''

In this picture, α represents the root of the subtree and $\alpha\alpha'$ (resp., $\alpha\alpha''$) represents the vertex v (resp., w). Without loss of generality, we assume $|\alpha'|{\leq}|\alpha''|$. Since the lowest common ancestor of v and w is at distance $\leq d$ from both vertices, we get $|\alpha''|{\leq}kd$. The definition of the mapping from the (k,h,o)-tree to the hypercube implies that $|\beta''|{\leq}k$, $|\delta''|{=}o$, and that γ' and γ'' contain exactly one 1 each. Hence, the labels of the two hypercube vertices differ in at most $kd{+}k{+}1{+}1{+}o{=}k(d{+}1){+}o{+}2$ positions.

We now consider the case that the vertex corresponding to v belongs to the set T, implying that v is in the image of the root of the (k,h,o)-tree. The vertex corresponding to w, labeled $\alpha'\beta'\gamma'\delta'$, belongs again to S. Since the distance between v and w is d, we have $|\alpha'|{\leq}kd$. Hence there are at most $kd{+}k{+}1{=}k(d{+}1){+}1$ 1's in the first kh 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 kh positions. Thus, the labels of these two hypercube locations differ in at most $k(d{+}1){+}1{+}1{+}o{=}k(d{+}1){+}o{+}2$ positions.

Finally, if both vertices belong to the set T, their labels obviously differ in at most $o{+}2$ positions. ■

```
1  Preprocessing
   1.1  Transform the decomposition tree to a binary decomposition tree
   1.2  Reorganize the binary decomposition tree
   1.3  Compute the inorder numbering of the reorganized binary decomposition tree
2  Main Algorithm
   2.1  Fill up the root of the $(k,h,o)$-tree
   2.2  Loop:
        2.2.1  Assign partition and marked vertices to the corresponding $(k,h,o)$-tree 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 $2^k$ ‘balanced’ parts
   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 1: The Embedding Algorithm

3 The Embedding

In this section, we give an embedding of a graph represented by a tree-decomposition into its optimal hypercube. First, we give an overview of the algorithm. Next, we explain in detail how to decompose a graph represented by a decomposition tree, which is the essential step in our algorithm. Finally, we give an estimation for the dilation and node-congestion of the embedding.

3.1 The Algorithm

Given a graph $G{=}(V,E)$, we denote the number of vertices in G by n, and the maximal degree of a vertex in G by d. By Lemma 1 and Lemma 2, we assume for simplicity that we have a binary tree-decomposition $\mathcal{D}_G{=}(S,F)$ of width t and size $\leq 2n$ for G. We will see that it is not necessary to bound the size of the tree-decomposition to obtain the claimed dilation.

In the following, we assume that $d{\geq}3$, since all graphs whose vertices have degree at most two can be embedded one-to-one into the 2-dimensional grid of size $2{\times}\lceil\frac{n}{2}\rceil$ with dilation two, which is obviously a subgraph of the optimal hypercube.

Our embedding of graphs with bounded treewidth into optimal hypercubes is achieved in two steps. First, we embed the graph into a (k,h,o)-tree. This will be explained in detail in the following. Then, we use the mapping presented in the previous section to complete the embedding.

To obtain a small dilation, adjacent vertices of the graph are mapped to nodes which are close in the (k,h,o)-tree. Our goal is to obtain an embedding of the graph into a (k,h,o)-tree such that adjacent vertices are mapped to two nodes of the (k,h,o)-tree with distance at most 1 from their lowest common ancestor of the (k,h,o)-tree. Our method leads to an embedding of a graph G into the hypercube with dilation $2k{+}o{+}2$.

The embedding of the graph G into the (k,h,o)-tree will be performed as follows. First, we fill up the root of the (k,h,o)-tree with arbitrarily chosen vertices of G and remove these vertices from G, yielding G'. Then, we mark the unmapped neighbors of the mapped vertices in the resulting graph G'. We associate this graph G' with the root of the the (k,h,o)-tree. Now we decompose this graph G' into 2^k parts using its decomposition tree such that the marked vertices are distributed evenly over all 2^k parts and that the number of edges between different parts is not too large. Such a decomposition will be given in the next subsection. Associate to each of the children one of the parts of the decomposed graph G'. We will call edges between different parts *partition edges* and vertices incident to such edges *partition vertices*.

In the next step, we fill up the children of the root with the marked vertices in the associated subgraph of G'. Additionally, we map the partition vertices of the previous decomposition to the 2^k children of the root. Finally, we fill up the nodes of the (k,h,o)-tree with arbitrarily chosen vertices of the associated subgraph until the capacity of this node exceeds.

We will repeat this process until we have reached the leaves of the (k,h,o)-tree. In this case, we map all vertices which belongs to the associated subgraph to the leaves of the (k,h,o)-tree.

Since we are looking for an embedding into the optimal hypercube, we have to choose the parameters of the (k,h,o)-tree such that $kh{+}o{=}\lceil\log(n)\rceil$. It will be shown that we can choose $k{=}\lceil\log(2(d{+}1)(t{+}1))\rceil$. Hence, we have to choose o such that $o{\equiv}\lceil\log(n)\rceil \bmod k$. Thus, the height of the (k,h,o)-tree is determined by $h{=}\frac{\lceil\log(n)\rceil-o}{k}$.

3.2 Partitioning a Graph with Bounded Treewidth

In this subsection, we present a technique to decompose a graph with bounded treewidth based on its decomposition tree. We start with a lemma on partitioning strings.

Let $s{\in}\{1,\ldots,m\}^*$. By $|s|_i$ we denote the number of occurrences of the symbol i in s, and by $|s|$ we denote the length of the string s. The following bisection lemma can be found in [GW85, AW86, BL84, Chu90].

Lemma 14 *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 $\lfloor\frac{|s|_k}{2}\rfloor{\le}\sum_{j\in J_\ell}|s_j|_k{\le}\lceil\frac{|s|_k}{2}\rceil$ and $\lfloor\frac{|s|}{2}\rfloor{\le}\sum_{j\in J_\ell}|s_j|{\le}\lceil\frac{|s|}{2}\rceil$ 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 $\lfloor\frac{|s|_1}{2}\rfloor$. The number of 2's in the window then is $\frac{n}{2}{-}\lfloor\frac{|s|_1}{2}\rfloor{=}\frac{|s|_1+|s|_2}{2}{-}\lfloor\frac{|s|_1}{2}\rfloor{=}\lceil\frac{|s|_2}{2}\rceil$. The two endpoints of the window give the required cuts for the strings. ■

We will need this proof later to construct such a bisection of a string s. Note that Lemma 14 is optimal in the sense that there exist strings that cannot be divided as required using fewer substrings.

Since a vertex in the decomposition tree represents several vertices of the given graph $G{=}(V, E)$, we introduce a *weight* function $w : S{\rightarrow}\mathbb{N}$ on the decomposition tree $T{=}(S, F)$ for G. For our purposes, the following weight function is sufficient (but another one could improve the dilation). For a tree vertex s, let $w(s){=}|X_s \backslash X_{\text{par}(s)}|$ except for the the root r of T, and let $w(r){=}|V_r|$, where par(s) denotes the parent of a vertex s in a tree. The weight of a subtree is defined as the sum of the weights of vertices which belong to this subtree. Note that $w(T){=}|V|$. We remark that condition *iii)* in the definition of a tree-decomposition and our definition of the weight function implies that subtrees with weight 0 can be removed from the decomposition tree without removing some edge of the represented graph.

As described in the previous subsection, we also have marked vertices in the subgraph associated to a (k, h, o)-tree node. We recall that a vertex is marked if at least one neighbor is mapped to a (k, h, o)-tree node in a previous stage. The marked graph vertices will be represented by a second weight function w' on the decomposition tree. For each vertex s in the decomposition tree, let $w'(s)$ be the number of marked vertices in $X_s \backslash X_{\text{par}(s)}$. Again, $w'(T)$ is equal to the number of marked vertices in G.

In order to partition decomposition trees using the above partition lemma, we need a linear representation of such forests like inorder strings. In the following, the string listing the vertices of T in inorder is called the inorder string of T. Because each vertex of the decomposition tree represents several vertices of the given graph G, we mean by a inorder string of a decomposition tree its inorder string, where each vertex s of the decomposition tree is replaced by the string $1^{w(s)-w'(s)}2^{w'(s)}$. Obviously, there is a one-to-one correspondence between the marked (resp., unmarked) vertices in the graph G and the 2's (resp., 1's) in the modified inorder string of the decomposition tree for G.

Since we are interested in partitioning a decomposition tree 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 string. 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 weight of its left (resp., right) subtree is grater equal than the weight of its right (resp., left) subtree. In the remainder of this paper, we refer to this procedure simply as the *reorganization* and call the resulting tree the *reorganized* tree. An efficient implementation of the reorganization on the hypercube is given in Section 4.3.

After the reorganization, a path from the root to a vertex with positive weight can use at most $\lfloor \log(w(T)) \rfloor$-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 weight of the remaining subtree.

This construction together with Lemma 14 can be used to partition a graph containing marked vertices represented by a binary decomposition trees into 2^k parts. But before we state the lemma on partitioning graphs represented by decomposition trees, we first need a technical lemma (see, e.g., [GKP]):

Lemma 15 *Let $f{:}\mathbb{R}{\to}\mathbb{R}$ be a continuous, monotonically increasing function with the following property: $f(x){\in}\mathbb{Z} \Rightarrow x{\in}\mathbb{Z}$. Then we have $\lceil f(\lceil x\rceil)\rceil{=}\lceil f(x)\rceil$.*

Proof: If $x{=}\lceil x\rceil$, there is nothing to prove. Otherwise $x{<}\lceil x\rceil$, and $f(x){\le}f(\lceil x\rceil)$ since f is monotonically increasing. Hence $\lceil f(x)\rceil{\le}\lceil f(\lceil x\rceil)\rceil$, since $\lceil\cdot\rceil$ is nondecreasing. If $\lceil f(x)\rceil{<}\lceil f(\lceil x\rceil)\rceil$, there must be a $y{\in}\mathbb{R}$ such that $x{\le}y{<}\lceil x\rceil$ and $f(y){=}\lceil f(x)\rceil$, since f is continuous. Because of the property of f, $y{\in}\mathbb{Z}$. But there cannot be an integer strictly between x and $\lceil x\rceil$. Hence we get a contradiction. ■

Now we are ready to prove the following lemma:

Lemma 16 *Let $G{=}(V,E)$ be a graph and $\mathcal{D}_G{=}(T,X)$ a tree-decomposition of width t for G and let $V'{\subseteq}V$ be a set of marked vertices in G. Then G can be decomposed into 2^k subgraphs $G_i{=}(V_i,E_i)$ ($i{\in}\{1,\ldots,2^k\}$) represented by 2^k decomposition trees T_i of width at most t such that the following conditions are satisfied:*

i) $\displaystyle\bigcup_{i=1}^{2^k} V_i = V$,

ii) $\displaystyle\bigcup_{i=1}^{2^k} E_i \subseteq E$,

iii) $|V_i|{\le}\left\lceil\dfrac{|V|}{2^k}\right\rceil$,

iv) in each G_i are at most $\left\lceil\dfrac{4(t+1)(d+1)(2^k-1)\log(|V|)+|V'|}{2^k}\right\rceil$ partition and marked vertices.

Proof: Proof by induction on k. Let $T{=}(S,F)$ be the decomposition tree for G and let $w,w'{:}S{\to}\mathbb{N}$ be two weight functions defined as above.

Induction base ($k{=}1$):
We first reorganize the decomposition tree T as described above. As mentioned earlier, we replace each vertex $s{\in}S$ in the inorder string of the reorganized decomposition tree by the string $1^{w(s)-w'(s)}2^{w'(s)}$. Using Lemma 14 for $m{=}2$ on the modified inorder string of the reorganized decomposition tree, we cut the inorder string at most twice. 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. Since we have replaced a vertex s of a decomposition tree by a string x of length $w(s)$, it

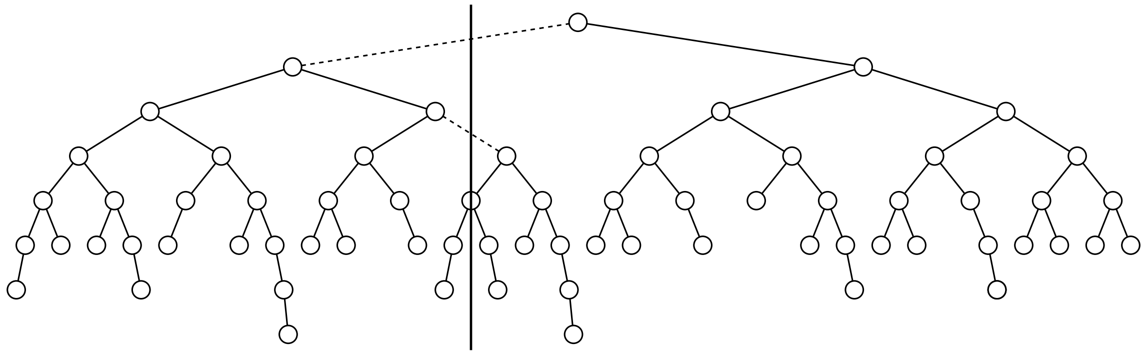


Figure 2: Decomposition of a Tree by Cutting its Inorder String

is possible that the cut through the inorder string could also cut the string x which means that a tree vertex could be cut too (cf. Figure 2). It can easily be seen that a cut through the modified inorder string of the decomposition tree can cut at most one tree vertex. If we cut a tree vertex of the decomposition tree, each part get a copy of this cut vertex. This yields two forests of decomposition trees which can be extended to two decomposition trees T_1 and T_2 of width at most t by adding some tree edges. The graphs G_i are the graphs represented by the decomposition trees T_i for $i{=}1,2$. Obviously, *i)* and *ii)* hold.

As stated above, at most $2\lceil\log(w(T))\rceil$ edges and vertices of the decomposition tree T could be cut. Consider a cut tree edge (s,u). Since only vertices in the set $X_s{\cap}X_u$ and their neighbors could be partition vertices, the number partition vertices produced by cutting a tree edge is at most $(t{+}1)(d{+}1)$. Cutting a vertex of the decomposition tree produces also at most $(t{+}1)(d{+}1)$ partition vertices. Hence at most

$$\left\lceil 2(t{+}1)(d{+}1)\log(w(T)){+}\frac{w'(T)}{2}\right\rceil = \left\lceil 2(t{+}1)(d{+}1)\log(|V|){+}\frac{|V'|}{2}\right\rceil$$

partition and marked vertices could be in each part G_i, which is claimed in condition *iv)*.

Condition *iii)* is obviously satisfied by Lemma 14.

Induction step ($k{-}1{\to}k$):
Using the induction hypotheses for $k{=}1$, we obtain a partition into two graphs G^1 and G^2 represented by two decomposition trees such that $|V^i|{\le}\lceil\frac{|V|}{2}\rceil$. Each G^i contains at most $\lceil 2(t{+}1)(d{+}1)\log(|V|)+\frac{|V'|}{2}\rceil$ marked and partition vertices.

Now, we also mark all partition vertices. Again, we reorganize both decomposition trees and modify the inorder string as above. Using on each graph G^i the induction hypotheses for $k{-}1$, we obtain a decomposition of G into 2^k graphs G_i represented by 2^k decomposition trees T_i of width t. Clearly, conditions *i)* and *ii)* hold.

The induction hypotheses and Lemma 15 implies that *iii)* is fulfilled:

$$|V_i| \le \left\lceil \frac{1}{2^{k-1}}\left\lceil\frac{|V|}{2}\right\rceil\right\rceil = \left\lceil\frac{|V|}{2^k}\right\rceil.$$

Using Lemma 15, the number of marked and partition vertices in each G_i can be bound by

$$\begin{aligned} w'(T_i) &\leq \left\lceil \frac{4(t+1)(d+1)(2^{k-1}-1)\log\left(\frac{|V|}{2}\right) + \left\lceil \frac{4(t+1(d+1))\log(|V|)+|V'|}{2} \right\rceil}{2^{k-1}} \right\rceil \\ &\leq \left\lceil \frac{4(t+1)(d+1)(2^k-2)\log(|V|) + 4(t+1)(d+1)\log(|V|) + |V'|}{2^k} \right\rceil \\ &\leq \left\lceil \frac{4(t+1)(d+1)(2^k-1)\log(|V|) + |V'|}{2^k} \right\rceil \end{aligned}$$

■

3.3 Dilation of the Embedding

It remains to show that we never map more vertices of the graph to a node of the (k, h, o)-tree as its capacity allows. Let $n(l)$ denote the maximal number of marked and partition vertices mapped to a node of the (k, h, o)-tree at level l. Hence, we have to show that $n(l){\leq}c(l)$ for all $1{\leq}l{\leq}h$. By $s(l)$ we denote the maximal size of the subgraph which is associated to a node of the (k, h, o)-tree at level l. Since we embed the graph $G{=}(V, E)$ into its optimal hypercube, we have $n{=}|V|{\leq}2^{kh+o}$. In each iteration the size of the associated subgraph to a node of the (k, h, o)-tree shrinks by 2^k so that $s(l){\leq}\left\lceil \frac{2^{kh+o}}{2^{k(l-1)}} \right\rceil$. Since we fill up the root of the (k, h, o)-tree with arbitrary nodes of the graph G, the inequality for $l{=}1$ holds. On the other hand, we can never map more than $c(h)$ vertices to a leaf of the (k, h, o)-tree, since in each stage we distribute the unmapped vertices of the graph evenly to the children of the considered node of the (k, h, o)-tree.

Now we bound for $2{\leq}l{\leq}(h{-}1)$ the number of partition and marked vertices mapped to a node of the (k, h, o)-tree. Let $G_i{=}(V_i, E_i)$ associated to a node x of the (k, h, o)-tree at level $l{-}1$. By Lemma 16 each graph associated to a child of x, and thus at level l, contains at most $\lceil \frac{1}{2^k}(4(t{+}1)(d{+}1)(2^k{-}1)\log(|V_i|){+}|V_i'|\rceil$ marked and partition vertices. Obviously, $|V_i|{\leq}s(l{-}1)$. The number of marked vertices of a graph associated to a (k, h, o)-tree node at level $l{-}1$ is at most $d{\cdot}c(l{-}1)$. Hence, we get $|V_i'|{\leq}d{\cdot}c(l{-}1)$. Altogether, we get the following bound on $n(l)$ which should be less equal $c(l)$.

$$\begin{aligned} n(l) &\leq \left\lceil \frac{d\cdot c(l-1) + 4(t+1)(d+1)(2^k-1)\lceil\log(s(l-1))\rceil}{2^k} \right\rceil \qquad (1) \\ &\leq \left\lceil \frac{d}{2^k}\left[2^o(2^k-1)k(h-l+2) - 2^o(2^k-1)(k-1) - k2^o + \delta_{l,2}2^o(hk+1)\right] \right\rceil \\ &\quad + 4\left\lceil \log\left(\frac{2^{kh+o}}{2^{k(l-2)}}\right) \right\rceil (t+1)(d+1) + 1 \\ &\leq 2^o d \left[k(h-l+2) - k + 1 + \delta_{l,2}\frac{hk+1}{2^k} \right] \\ &\quad + 4(t+1)(d+1\left[kh+o-kl+2k\right]+1 \end{aligned}$$

$$\begin{aligned}&\leq \quad 2^o d\left[k(h-l+1)+1+\delta_{l,2}\frac{hk+1}{2^k}\right]\\ &\qquad +4(t+1)(d+1)\left[k(h-l+1)+o+k\right]+1\\ &\overset{!}{\leq} \quad c(l)=2^o(2^k-1)k(h-l+1)-2^o(2^k-1)(k-1)-2^o k\end{aligned}$$

The last inequality is certainly true if the following inequalities are true:

$$2^o d+4(t+1)(d+1)(o+k)+2^o(2^k-1)(k-1)+2^o k+1+\delta_{l,2}2^o\frac{d(hk+1)}{2^k}$$
$$\overset{!}{\leq} \quad (h-l+1)k\left[2^o(2^k-1)-2^o d-4(t+1)(d+1)\right]$$
$$\Rightarrow$$
$$2^o d+4(t+1)(d+1)(o+k)+2^o2^k k-2^o2^k+2^o+1+\delta_{l,2}2^o\frac{d(hk+1)}{2^k}$$
$$\overset{!}{\leq} \quad (h-l+1)k\left[2^o(2^k-d-1)-4(t+1)(d+1)\right]$$
$$\Rightarrow$$
$$2^o\left(d-2^k+1+\frac{1}{2^o}\right)+2^o2^k k+4(k+o)(t+1)(d+1)+\delta_{l,2}2^o\frac{d(hk+1)}{2^k}$$
$$\overset{!}{\leq} \quad (h-l+1)k\left[2^o(2^k-d-1)-4(t+1)(d+1)\right]$$
$$\Rightarrow$$
$$2^o\left(d+1+\frac{1}{2^o}-2^k\right)+2^o2^k k+4(k+o)(t+1)(d+1)+\delta_{l,2}2^o\frac{d(hk+1)}{2^k}$$
$$\overset{!}{\leq} \quad (h-l+1)k\left[2^o(2^k-d-1)-4(t+1)(d+1)\right]$$

As mentioned above, we now choose $k{=}\lceil\log(2(d{+}1)(t{+}1))\rceil$ and $o{\in}[4{:}k{+}3]$ such that $o{\equiv}\lceil\log(n)\rceil \bmod k$. Thus, we get:

$$\begin{aligned}&2^o\left((d+1)+\frac{1}{2^o}-2(d+1)(t+1)\right)\\ &+2\cdot 2^o\cdot k(t+1)(d+1)+4(k+o)(t+1)(t+1)+\delta_{l,2}2^o\frac{d(hk+1)}{2(d+1)(t+1)}\\ &\qquad \overset{!}{\leq} \quad (h-l+1)k\left[2^o(2(t+1)(d+1)-(d+1))-4(t+1)(d+1)\right]\end{aligned}$$

$\Rightarrow$ (Division by k)

$$(d+1)\frac{2^o}{k}+\frac{1}{k}+(t+1)(d+1)\left[2\cdot 2^o\left(1-\frac{1}{k}\right)+4\frac{k+o}{k}\right]+\delta_{l,2}2^o\frac{h+1}{2(t+1)}$$
$$\overset{!}{\leq} \quad (h-l+1)\left[(t+1)(d+1)(2\cdot 2^o-4)-2^o(d+1)\right] \qquad (2)$$

For $3{\leq}l{\leq}h{-}1$, the above inequality (2) is satisfied if the following inequality holds (note, that $o{\geq}4$ and $(h{-}l{+}1){\geq}2$):

$$\frac{2^o}{k}+\frac{1}{k(d+1)}+(t+1)\left[2\cdot 2^o(1-\frac{1}{k})+8+\frac{12}{k}\right]$$
$$\overset{!}{\leq} \quad (t+1)(4\cdot 2^o-8)-2\cdot 2^o$$

$\Rightarrow$

$$2 \cdot 2^o + \frac{2^o}{k} + \frac{1}{k(d+1)} \overset{!}{\leq} (t+1)\left[2 \cdot 2^o\left(1+\frac{1}{k}\right) - 16 + \frac{12}{k}\right]$$

Since $t \geq 1$, the following inequality should be satisfied:

$$32 + \frac{1}{k(d+1)} \overset{!}{\leq} 2 \cdot 2^o + \frac{3}{k}2^o + \frac{24}{k}$$

Because $o \geq 4$ and $k \geq 4$ this inequality is true.

For $l=2 \leq h-1$, from inequality (2) follows that the following inequality should be satisfied.

$$(d+1)\frac{2^o}{k} + \frac{1}{k} + (t+1)(d+1)\left[2 \cdot 2^o\left(1-\frac{1}{k}\right) + 4\frac{k+o}{k}\right] + 2^o\frac{h-1+2}{2(t+1)} \overset{!}{\leq} (h-1)\left[(t+1)(d+1)(2 \cdot 2^o - 4) - 2^o(d+1)\right]$$

$\Rightarrow$ (Division by $(d+1)2^o$)

$$\frac{1}{k} + \frac{1}{2^o k(d+1)} + (t+1)\left[2\left(1-\frac{1}{k}\right) + 4\frac{k+o}{k2^o}\right] + \frac{1}{(t+1)(d+1)} \overset{!}{\leq} (h-1)\left[(t+1)\left(2-\frac{4}{2^o}\right) - 1 - \frac{1}{2(d+1)(t+1)}\right]$$

$\Rightarrow$ (by $k \geq 4$, $o \geq 4$, $d \geq 3$ and $t \geq 1$)

$$\frac{(t+1)\left[2+\frac{4+o}{2^o}\right] + \frac{1}{2}}{(t+1)\left[2-\frac{4}{2^o}\right] - \frac{17}{16}} \overset{!}{\leq} (h-1) \qquad (3)$$

For each fixed $t \geq 1$, the left hand side of inequality (3) converges as $o \to \infty$. It can be easily verified from Figure 3 that for $o \geq 4$ the left hand side of inequality (3) is less then 2 except for $(t=1, o=4)$. A simple recomputation of the right hand side of inequality (1) for $h=3$, $t=1$, and $o=4$ yields that it is less than $c(2)$ for $d \geq 3$. Hence, for $h \geq 3$ we obtain $n(l) \leq c(l)$ for all $l \in [2{:}h]$. This implies the required bound on the dilation for our embedding. For $h \leq 2$, we have trivially a dilation $2k+o$ embedding of any graph with at most 2^{2k+o} vertices into its optimal hypercube.

A careful inspection of the previous estimation shows that the dilation of the embedding does not depend on the size of the given decomposition tree. As we will see in the following section, only the running time of the algorithm depends on the size of the given decomposition tree.

Altogether, we obtain the following theorem.

Theorem 17 *Let $G=(V,E)$ be a graph with treewidth t and maximal degree d. There exists a one-to-one embedding of G into its optimal hypercube with dilation at most $3\lceil \log(d+1)(t+1) \rceil + 8$.*

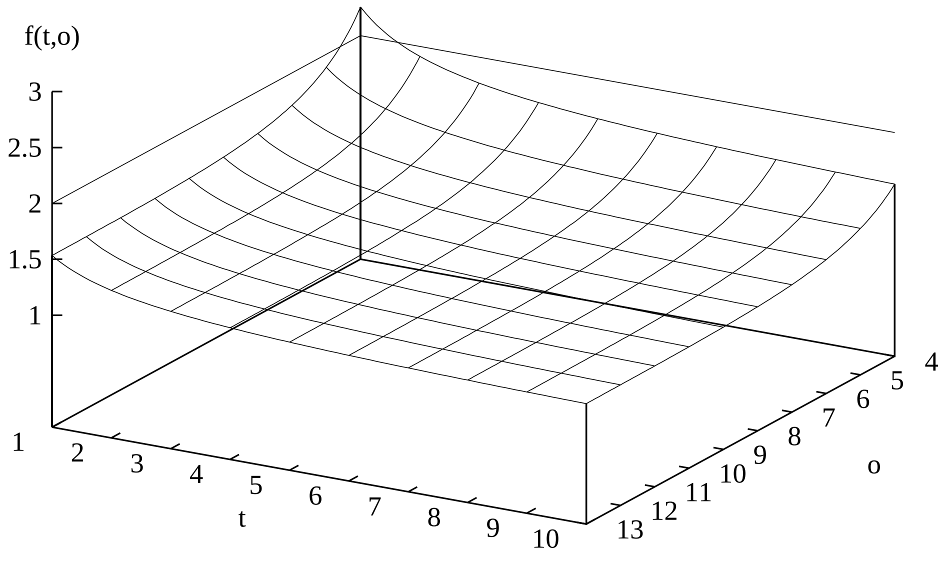


Figure 3: Graph of $f(t,o)=\frac{(t+1)[2+\frac{4+o}{2^o}]+\frac{1}{2}}{(t+1)[2-\frac{4}{2^o}]-\frac{17}{16}}$

Proof: We have shown that we can embed such a graph one-to-one into its optimal hypercube with dilation $2k+o+2$, where $k=\lceil\log(2(d+1)(t+1))\rceil$ and $o\geq 4$ such that $o\equiv\lceil\log(n)\rceil \bmod k$. Thus, we have $o\leq k+3$. Altogether, we get an an upper bound for the dilation of $3k+5\leq 3\lceil\log(d+1)(t+1)\rceil+8$. ■

3.4 Congestion of the Embedding

We restrict our attention to bound the 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 o 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 4 for the case $v,w\in S$). Also note that segments A and B can be empty.

For a path $p_{v,w}$ from v to w, we call v the *lower* endpoint of $p_{v,w}$, if $v=\alpha_v\beta_v\gamma_v\delta_v\in S$, $w=\alpha_w\beta_w\gamma_w\delta_w\in S$ (cf. Figure 5) 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.

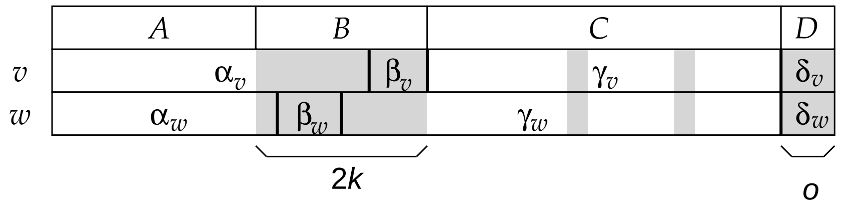


Figure 4: Hypercube Locations of Two Adjacent Tree Vertices

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 graph edge. If this number is bounded by c, the node-congestion is bounded by $c+d$, since the degree of a vertex is at most d.

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 kh positions. While flipping bits in segments B and D, u and the label of the lower endpoint agree except possibly in the shaded segments in row a) of Figure 5. Since the length of the shaded segments is bounded by $2k$, at most 2^{2k} paths can hit u during the first phase.

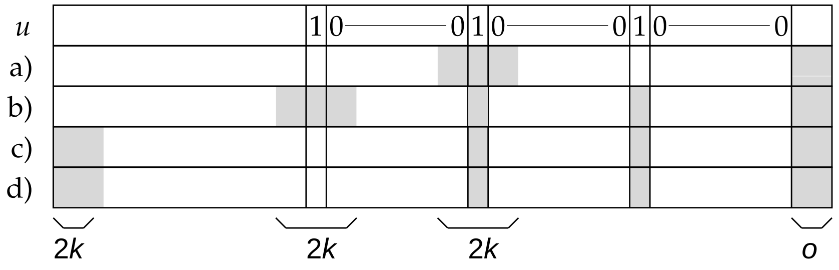


Figure 5: Possible Lower Endpoints of a Path Hitting u

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 5. Again, since the length of the shaded segments is bounded by $2k$, at most 2^{2k} 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 $2k$ 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 5 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 5. Thus, there again exists at most 2^{2k} paths hitting u.

Altogether, we have shown that for any hypercube location u at most a $O((d{+}t)^2)$ paths can hit u, finishing the proof.

Theorem 18 *Let $G{=}(V,E)$ be a graph with treewidth t and maximal degree d. There exists a one-to-one embedding of G into its optimal hypercube with dilation at most $3\lceil\log(d{+}1)(t{+}1)\rceil{+}8$ and node-congestion $O((d{+}t)^2)$.* $\square$

4 Implementation on the Hypercube

In this section, it will be shown that the algorithm of the previous section can be efficiently implemented on the optimal hypercube. This section is organized as follows. First, we present some fundamental hypercube algorithms, which we will need in the following. Next, we discuss some reasonable restrictions to obtain an efficient implementation. Finally, we describe the preprocessing step, the main algorithm, and a procedure for decomposing a graph represented by a tree-decomposition.

4.1 Fundamental Hypercube-Algorithms

First, we mention that we call in the following vertices in the hypercube processors. In this subsection, we consider a hypercube of size n.

Let $H{=}(M,\circ)$ be a semigroup. Given at each processor $i{\in}\{0,\ldots,n{-}1\}$ of the hypercube an element $m_i{\in}M$, we can compute at each processor i of the hypercube the value $\bigcirc_{j=0}^{i} m_i$ in time $O(\log(n))$ (see [Sch80]). This is called a *parallel prefix computation*. Sometimes we are interested in the postfix sum instead of the prefix sum. Clearly, this is computationally equivalent.

A variation of the parallel prefix computation is the *segmented parallel prefix computation*. Additionally, we have a sequence $0{=}j_0{<}\cdots{<}j_m{<}n$ of $m{\leq}n$ integers, where each j_l is stored at processor j_l as a marker. On the hypercube we can compute at each processor i the values $\bigcirc_{j=\max\{j_l\leq i\}}^{i} m_i$ in time $O(\log(n))$ (see [Sch80]). It can be shown that the segmented parallel prefix problem can be reduced to a parallel prefix problem (see, e.g., [Lei92, MP93]).

Performing a series of l independent parallel prefix or segmented parallel prefix computations can implemented using pipelining in time $O(l{+}\log(n))$. We note that various broadcast operations can be implemented using parallel prefix or segmented parallel prefix computations.

Concentrating $m{\leq}n$ elements stored one element per processor into a contiguous block of m hypercube locations preserving their order can be implemented in time $O(\log(n))$ (see [NS81]). We call this algorithm *concentration routing* and the inverse operation *inverse concentration routing*. If the elements should be stored in reverse order we call it *reverse concentration routing*. This can be implemented in time $O(\log(n))$ too.

Another fundamental special case of permutation routing is *monotone routing*: provided that each processor is the source and the destination of at most one element, the problem is to route the elements preserving their order from their source to their destination processor. This can be computed by a concentration routing followed by a inverse concentration routing. Hence, monotone routing can be solved in time $O(\log(n))$.

Let $\mathrm{T}_{\mathsf{Sort}}(m,d)$ denote the time required to sort m items on a d-dimensional hypercube where at most one item is stored per processor. The sorting algorithm given in [CP90] yields $\mathrm{T}_{\mathsf{Sort}}(m,\lceil\log(m)\rceil){=}O(\log(m)\,\mathrm{loglog}^2(m))$. For sorting a small number of items, it was shown in [NS82] that for sparse enumeration sort we get $\mathrm{T}_{\mathsf{Sort}}(m,d){=}O\left(\frac{d\log(m)}{d-\log(m)+1}\right)$. Note that these sorting algorithms are also the fastest known online algorithms for (partial) permutation routing. In the following, we write $\mathrm{T}_{\mathsf{Sort}}(m)$ for $\mathrm{T}_{\mathsf{Sort}}(m,\lceil\log(m)\rceil)$.

Given a linked list of length n, *list ranking* is the problem to determine for each item in the list the number of its successors in the list. We denote by $\mathrm{T}_{\mathsf{LR}}(n)$ the time for solving the list ranking problem on a $\lceil\log(n)\rceil$-dimensional hypercube for a list of length n stored one item per processor. The fastest known algorithm, given in [HM93], requires time $\mathrm{T}_{\mathsf{LR}}(n){=}O(\log^2(n)\,\mathrm{logloglog}(n)\log^*(n))$.

We also will 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 $O(\max\{\mathrm{T}_{\mathsf{Sort}}(n),\mathrm{T}_{\mathsf{LR}}(n)\})$.

If each processor contains $l{\leq}n^2$ items instead of one item, the algorithms mentioned above can be implemented on a hypercube of size n with a slowdown of l.

4.2 Preliminary Remarks

In the rest of this section, we assume that the graph G has size n, treewidth $\leq t$, maximal degree d, and that the size of the given decomposition tree for G is less than n. Our algorithm can be easily modified to work with decomposition trees of size $N{>}n$; this yields a slow-down of $\frac{N}{n}$. As mentioned above, we will give our implementation on the optimal hypercube, i.e. a hypercube of dimension $\lceil\log(n)\rceil$.

In the following, we assume that at most one vertex of the decomposition tree is stored per processor. The edges of the decomposition tree are given as a link par(v) of each vertex v to its parent except for the root. The set X_s and the information about edges in G between vertices in X_s are also stored in the processor containing s. Whenever a vertex s of the decomposition tree is remapped to another processor, the set X_s and the information whether two vertices in X_s are adjacent will be remapped too. Hence, we have to perform

```
1 Transform the decomposition tree into a binary decomposition tree
2 For each vertex of the binary decomposition tree, compute the weight of its subtree, its depth,
  and its inorder number; also compute the height of the tree
3 For each vertex of the binary decomposition tree, compute the following information about its
  neighborhood in the binary decomposition tree:
    3.1 the inorder numbers for its children and its parent
    3.2 the size of the subtrees of its children and grandchildren
4 Reorganize the binary decomposition tree
5 Compute the new inorder numbering of the reorganized binary decomposition tree
6 Route each vertex $s$ of the decomposition tree together with the set $X_s$ to the location in the
  hypercube given by its inorder number
7 Repeat step 3 for the reorganized binary decomposition tree
```

Figure 6: Preprocessing Stage of the Hypercube Algorithm

a permutation routing to remap this information which requires time $O(t^2 \cdot \mathrm{T}_{\mathsf{Sort}}(n))$. All other reasonable representations can be converted to this representation on the hypercube in time $O(\max\{t^2 \cdot \mathrm{T}_{\mathsf{Sort}}(n), \mathrm{T}_{\mathsf{LR}}(n)\})$.

In Lemma 16, we have constructed a decomposition of a graph using its tree-decomposition. Differently from the proof of Lemma 16, we do not add edges to obtain a new decomposition tree. Instead, we will work with a forest of binary decomposition trees.

4.3 Preprocessing

First, we describe the transformation of the decomposition tree into a binary one. To determine the in-degree of each vertex, we sort the tree vertices by the value par(v). Using two segmented parallel prefix operations, we can determine for each vertex the number of its siblings. Therefore, the degree of each vertex can be determined using a permutation routing. Now we can easily extend the tree-decomposition into a binary tree-decomposition as described in Lemma 2. For each new tree vertex s, we have to initialize X_s. Thus, step 1 of the preprocessing can be computed in time $O(t^2 \cdot \mathrm{T}_{\mathsf{Sort}}(n))$.

First, we note that the weight of each vertex can be computed in time $t \cdot \mathrm{T}_{\mathsf{Sort}}(n)$. To compute, for each vertex, the weight of its subtree in step 2 of the preprocessing, we mark LEFT(v) with $w(v)$ 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 weight of its subtree by subtracting the value computed at LEFT(v) from the value computed at RIGHT(v), and adding $w(v)$. 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. Altogether, step 2 can be implemented in time $O(t \cdot \mathrm{T}_{\mathsf{Sort}}(n) + \mathrm{T}_{\mathsf{LR}}(n))$.

Step 3 and step 7 consists of a constant number of routing problems which can all be solved using sorting. For step 4 of the preprocessing, we note that whether a vertex is

```
1 Fill the root of the $(k,h,o)$-tree up to capacity with arbitrarily chosen vertices and compute its
  hypercube location
2 Loop:
    2.1 Map the partition and marked vertices to the corresponding $(k,h,o)$-tree nodes
    2.2 Fill the $(k,h,o)$-tree 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 graph into $2^k$ parts of the nearly the same size using the procedure
        Decompose-Graph and divide the cube into $2^k$ subcubes and route each newly created
        forest of binary decomposition trees to its own subcube
3 Route the vertices to their computed locations in the hypercube and determine, for each vertex,
  the locations of its neighbors
```

Figure 7: Main Part of Hypercube Algorithm

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. As mentioned in the previous subsection, step 6 of the preprocessing can be implemented in time $O(t^2 \cdot \mathrm{T}_{\mathsf{Sort}}(n))$.

Altogether, we can implement the preprocessing in time $O(\max\{\mathrm{T}_{\mathsf{LR}}(n), t^2 \cdot \mathrm{T}_{\mathsf{Sort}}(n)\})$.

4.4 Main Part

The main algorithm is as stated in Figure 7. The first step can easily be implemented in time $O(t \cdot \log(n))$ using parallel prefix operations. Step 3 can solved in time $O((t+d) \cdot \mathrm{T}_{\mathsf{Sort}}(n))$.

Each iteration of the loop in step 2 performs the assignment of vertices to the nodes at the corresponding level of the (k,h,o)-tree. Thus, the l-th iteration is performed in $2^{k(l-1)}$ distinct $(k(h-l+1)+o)$-dimensional subcubes, where each subcube contains a forest of binary decomposition tree of size at most $\lceil \frac{n}{2^{k(i-1)}} \rceil$. For the following discussion, we consider a fixed iteration of the loop and we denote by m the size of the actual subcube. Note that $(h-l+1) \leq \frac{\log(m)}{k}$.

Clearly, step 2.1, step 2.2, and step 2.3 can be implemented in time $O(t + \log(m))$ using parallel prefix operations In step 2.4, we have to solve a broadcasting problem, since, for each mapped vertex, we have to mark its (unmapped) neighbors. By construction, the number of mapped vertices is at most $c(l-1) \leq O(k 2^k (h-l+2)) = O(2^k \log(m))$. In each forest of binary decomposition trees we broadcast which vertices are mapped in the last stage. This can be done in time $O(2^k \log(m)) = O((d+t)\log(m))$ using a pipelined segmented parallel prefix operation. Since in each tree vertex are stored at most t graph vertices, we can determine the subset of mapped vertices stored in that tree vertex in time $O((d+t)\log(t)\log(m))$. To decide whether a graph vertex should be marked, i.e. whether

```
1 Decompose the forest of binary decomposition trees into two forests, such that each newly
  created forest contains one half of the marked vertices as well as one half of all vertices of the
  graph to be embedded
2 Update the information about the size and weight of subtrees of the children and grandchildren:
    2.1 Compute and broadcast the size and weight of the removed subtree(s) in each tree
    2.2 Update the information about the size and weight of subtrees of its children and grand-
        children
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 weight of the left subtree became smaller than the weight of 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
```

Figure 8: Procedure Decompose-Graph

it is a neighbor of a mapped vertex, can be done in time $O(t^2)$. Altogether, step 2.4 requires time $O((d{+}t)\log(t)\log(m){+}t^2)$.

In the next subsection, we will show that the procedure Decompose-Graph can be implemented in time $O(k\cdot((d{+}t^2)\log(m)))$.

Since there are $h{=}O(\frac{\log(n)}{k})$ stages, altogether the time complexity of the main part of the algorithm is bounded by $O((d{+}t^2)\log^2(n))$.

4.5 Partitioning

In this subsection, we describe in detail how to decompose a graph represented by a binary decomposition tree in step 2.5 of the main algorithm. We will only show how to partition a graph represented by a forest of binary decomposition trees into two parts in time $O((d{+}t^2)\log(m))$, where m is the actual size of the used subcube. Applying this partitioning to each newly created part in parallel, we obtain after k iterations a partition into 2^k parts in time $O(k(d{+}t^2)\log(m))$. As noticed in subsection 4.2, after each bisection, the information about X_s and whether two vertices in X_s are connected have to be remapped together with s. This can be implemented in time $O(kt^2\log(m))$ using t^2 concentration routings, which implies that step 2.5 in the main part of the algorithm requires time $O(k(d{+}t^2)\log(m))$. The procedure Decompose-Graph is listed in Figure 8.

Using the method given in the proof of Lemma 14, the inorder string can be bisected in time $O(t{+}\log(m))$ using a parallel prefix computation and a monotone routing. It remains to determine which vertices of the given graph are partition vertices. A vertex v of the given graph is a partition vertex iff $v{\in}X_r{\cap}X_s{\cup}N(X_r{\cap}X_s)$ for some cut edge (r,s). Note, that there are at most $O(\log(m))$ cut edges. Using the information about the neighborhood of a decomposition tree vertex, it can be easily determined in parallel whether a decomposition tree vertex is incident to a cut edge. Hence, the vertices in $X_r{\cap}X_s$ for all cut edges (r,s) can be computed in time $O(t\log(m))$ by sending X_r to vertex s and vice verse using t

times sparse enumeration sort. Now, we broadcast which vertices belong to $X_r \cup X_s$ for all cut edges (r,s). Because there are at most $O(t\log(m))$ such vertices, this can be done using pipelined segmented parallel prefix in time $O(t\log(m))$. Next, the partition vertices can be determined in time $O(t^2)$. As described in subsection 4.4, the neighbors of the partition vertices can be marked in time $O((d+t^2)\log(m))$. Altogether, step 1 requires time at most $O((d+t^2)\log(m))$.

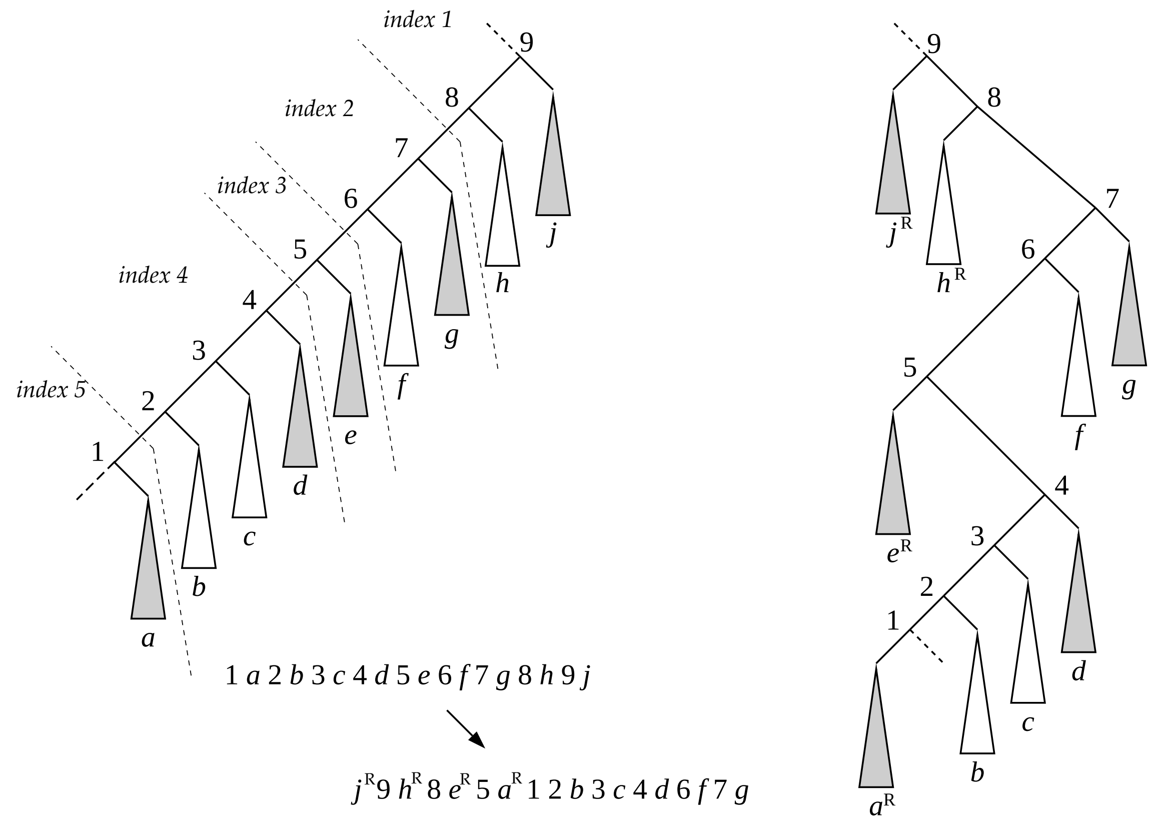
For simplicity, we now consider only one binary decomposition tree of the forest after the bisection, and we note that all computations are executed in parallel for all such trees. The operations required are parallel prefix computation, broadcasting, concentration routing, and reverse concentration 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 have 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 and weight of its subtree. Each vertex knows the size and weight 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 and weight of its subtree that was removed. Subsequently, each vertex on the trunk subtracts these values from the size and weight, respectively, 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 time $O(\log(m))$.

Next, the tree is reorganized as described in Section 3.2. After the partitioning, the property that the weight of the left (resp., right) subtree of a left (resp., right) vertex is larger than the weight of the right (resp., left) subtree may no longer hold. Call the corresponding vertices *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



index 1
index 2
index 3
index 4
index 5
9
8
7
6
5
4
3
2
1
a
b
c
d
e
f
g
h
j
1 a 2 b 3 c 4 d 5 e 6 f 7 g 8 h 9 j
j^R 9 h^R 8 e^R 5 a^R 1 2 b 3 c 4 d 6 f 7 g
j^R
h^R
e^R
a^R

Figure 9: Reorganization of a Tree

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 time $O(t^2 \log(m))$ using t^2 times algorithms for concentration and reverse concentration routing. Again, it is clear that time $O(t^2 \log(m))$ suffices for steps 3.1 and 3.2.

This part of the algorithm is illustrated in Figure 9, 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.

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. Thus, again the operations of step 4 require at most time $O(\log(m))$.

Altogether, we have proved the following theorem:

Theorem 19 *Let $G{=}(V,E)$ be a graph with treewidth t, maximal degree d and size n. Given a tree-decomposition for G of width t and size $O(n)$, there exists a one-to-one embedding of G into its optimal hypercube with dilation $3\lceil\log((d{+}1)(t{+}1))\rceil{+}8$ and node-congestion $O((d{+}t)^2)$. This embedding can be computed in time*

$$O\left(\max\left\{\mathrm{T}_{\mathsf{LR}}(n),(d+t^2)\log^2(n)\right\}\right)$$
$$= \begin{cases} O(\log^2(n)\,\mathrm{logloglog}(n)\log^*(n)) & \textit{if} \quad d+t^2 = O(\mathrm{logloglog}(n)\log^*(n)) \\ O((d+t^2)\log^2(n)\}) & \textit{if} \quad d+t^2 = \Omega(\mathrm{logloglog}(n)\log^*(n)) \end{cases}$$

on the optimal hypercube. □

5 Conclusion

We have shown an algorithm to achieve a one-to-one embedding of a graph with treewidth t and maximal degree d into its optimal hypercube with dilation $3\lceil\log((d{+}1)(t{+}1))\rceil{+}8$ and node-congestion at most $O((d{+}t)^2)$. This is the first time that an embedding of a class of highly irregular graphs into their optimal hypercube is presented. For graphs with constant treewidth and constant degree, our construction yields an embedding with constant dilation and constant node-congestion into the optimal hypercube. Moreover, this embedding can be efficiently computed on the optimal hypercube itself.

It remains open to study dynamic embeddings of graphs with bounded treewidth into hypercubes. 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 $\Omega((\log(n))^{\frac{1}{2}})$ in the worst case. This follows by a simple adaptation of a result in [LNRS89, LNRS92]. Hence, no deterministic dynamic embedding can achieve small dilation without migration.

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.

[ACP87] S. Arnborg, D. Corneil, A. Proskurowski: Complexity of Finding Embeddings in a k-tree, *SIAM Journal on Algebraic Discrete Methods*, Vol. 8 (1987), No. 2, 277–284.

[BMUW93] S. Bezrukov, B. Monien, W. Unger,G. Wechsung, Embedding Caterpillars into the Hypercube, Prepint, GH-Universität Paderborn, 1993.

[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 Journal on Computing*, Vol. 21 (1992), No. 1, 151–162.

[BL84] S. Bhatt, T. Leighton: A Framework for Solving VLSI Graph Layout Problems, *Journal on Computer and System Sciences*, Vol. 28 (1984), 300–334.

[BH95] H. Bodlaender, T. Hagerup: Parallel Algorithms with Optimal Speedup for Bounded Treewidth, Utrecht University, Technical Report UU-CS-1995-25, 1995.

[Cha89] M.Y. Chan: Embedding of d-Dimensional Grids into Optimal Hypercubes, *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 52–57.

[Cha91] M.Y. Chan: Embedding of Grids into Optimal Hypercubes, *SIAM Journal on Computing* Vol. 20 (1991), No. 5, 834–864.

[CCCM93] M.Y. Chan, F. Chin, C.N. Chu, W.K. Mak: Dilation-5 Embedding of 3-Dimensional Grids into Hypercubes *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing 1993*, 285–289.

[Chu78] F. Chung: On Concentrators, Superconcentrators, Generalizers and Nonblocking Networks, *Bell Sys. Tech. J. 58* (1978), 1765-1777. in *B. Korte, L. Lovász, H. J. Prömel, A. Schrijver: Paths, Flows, and VLSI-Layout*, Springer-Verlag, 1990, 17–34.

[Chu90] 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.

[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.

[Efe91] K. Efe: Embedding Mesh of Trees in the Hypercube, *Journal Parallel and Distributed Computing*, Vol. 11 (1991), 222–230.

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

[GKP] L. Graham, D.E. Knuth, O. Patashnik: Concrete Mathematics, A Foundation of Computer Science, Addison-Wesley, 1989.

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

[Hav84] I. Havel: On Hamiltonian Circuits and Spanning Trees of Hypercubes (in Czech.), *Časopis. Pěst. Mat.*, Vol. 109 (1984), 145–152.

[HL73] I. Havel, P. Liebl: Embedding the Polytomic Tree into the n-Cube, *Časopis. Pěst. Mat.*, Vol. 98 (1973), 307–314.

[HL86] I. Havel, P. Liebl: One-Legged Caterpillars Span Hypercubes, *Journal of Graph Theory*, Vol. 10 (1986), 69–76.

[HM93] V. Heun, E.W. Mayr: A New Efficient Algorithm for Embedding an Arbitrary Binary Tree into Its Optimal Hypercube, Technical Report, TUM-I9321, Technische Universität München, 1993 (to appear in *Journal of Algorithms*).

[Klo94] T. Kloks: Treewidth: Computations and Approximations, *Lecture Notes in Computer Science*, Vol. 842, Springer-Verlag, 1994.

[Lei92] 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 Architectures*, 224–234.

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

[MP93] E.W. Mayr, C.G. Plaxton: Pipelined Parallel Prefix Computations, and Sorting on a Pipelined Hypercube, *Journal of Parallel and Distributed Computing.*, Vol. 17 (1993), No. 4, 374–380.

[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.

[NS81] D. Nassimi, S. Sahni: Data Broadcasting in SIMD Computers, IEEE Transactions on Computers, Vol. C-30 (1981), 101-107.

[NS82] D. Nassimi, S. Sahni: Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network, *Journal of the ACM*, Vol. 29 (1982), No. 3, 642–667.

[Pin73] M. Pinsker: On the Complexity of a Concentrator, *7th International Teletraffic Conference* (1973), 318/1-318/4.

[Pip77] N. Pippenger: Superconcentrators, *SIAM Journal on Computing*, Vol. 6 (1977), 298-304.

[Sch80] J.T. Schwartz: Ultracomputers, *ACM Transactions on Programming Languages and Systems*, Vol. 2 (1980), 484–521.

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

[SHL95] X. Sheen, Q. Hu, W. Liang: Embedding k-ary Complete Trees into Hypercubes, *Journal of Parallel and Distributed Computing*, Vol. 24 (1995), 100–106.

[Sto86] Q. Stout: Hypercubes and Pyramids, *Proceedings of the NATO Advanced Research Workshop on Pyramidal Systems for Computer Vision 1986*, V. Cantoni, S.Levialdi, eds., Springer, Series F: Computers and System Science, 75–89.

[WC90] A. Wagner, D. Corneil: Embedding Tree in a Hypercube is NP-Complete, *SIAM Journal on Computing*, Vol. 19 (1990), No. 3, 570–590.

[WC93] A. Wagner, D. Corneil: On the Complexity of the Embedding Problem for Hypercube Related Graphs, *Discrete Applied Mathematics*, Vol. 43 (1993), 75-95.

Reihe A

342/20/90 A	Michael Griebel: A Parallelizable and Vectorizable Multi- Level-Algorithm on Sparse Grids
342/21/90 A	V. Diekert, E. Ochmanski, K. Reinhardt: On confluent semi-commutations-decidability and complexity results
342/22/90 A	Manfred Broy, Claus Dendorfer: Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions
342/23/90 A	Rob van Glabbeek, Ursula Goltz: A Deadlock-sensitive Congruence for Action Refinement
342/24/90 A	Manfred Broy: On the Design and Verification of a Simple Distributed Spanning Tree Algorithm
342/25/90 A	Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Peter Luksch, Roland Wismüller: TOPSYS - Tools for Parallel Systems (User's Overview and User's Manuals)
342/26/90 A	Thomas Bemmerl, Arndt Bode, Thomas Ludwig, Stefan Tritscher: MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual)
342/27/90 A	Wolfgang Ertel: Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems
342/28/90 A	Rob van Glabbeek, Frits Vaandrager: Modular Specification of Process Algebras
342/29/90 A	Rob van Glabbeek, Peter Weijland: Branching Time and Abstraction in Bisimulation Semantics
342/30/90 A	Michael Griebel: Parallel Multigrid Methods on Sparse Grids
342/31/90 A	Rolf Niedermeier, Peter Rossmanith: Unambiguous Simulations of Auxiliary Pushdown Automata and Circuits
342/32/90 A	Inga Niepel, Peter Rossmanith: Uniform Circuits and Exclusive Read PRAMs
342/33/90 A	Dr. Hermann Hellwagner: A Survey of Virtually Shared Memory Schemes
342/1/91 A	Walter Vogler: Is Partial Order Semantics Necessary for Action Refinement?
342/2/91 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Rainer Weber: Characterizing the Behaviour of Reactive Systems by Trace Sets
342/3/91 A	Ulrich Furbach, Christian Suttner, Bertram Fronhöfer: Massively Parallel Inference Systems
342/4/91 A	Rudolf Bayer: Non-deterministic Computing, Transactions and Recursive Atomicity
342/5/91 A	Robert Gold: Dataflow semantics for Petri nets
342/6/91 A	A. Heise; C. Dimitrovici: Transformation und Komposition von P/T-Netzen unter Erhaltung wesentlicher Eigenschaften
342/7/91 A	Walter Vogler: Asynchronous Communication of Petri Nets and the Refinement of Transitions
342/8/91 A	Walter Vogler: Generalized OM-Bisimulation

Reihe A

342/9/91 A	Christoph Zenger, Klaus Hallatschek: Fouriertransformation auf dünnen Gittern mit hierarchischen Basen
342/10/91 A	Erwin Loibl, Hans Obermaier, Markus Pawlowski: Towards Parallelism in a Relational Database System
342/11/91 A	Michael Werner: Implementierung von Algorithmen zur Kompaktifizierung von Programmen für VLIW-Architekturen
342/12/91 A	Reiner Müller: Implementierung von Algorithmen zur Optimierung von Schleifen mit Hilfe von Software-Pipelining Techniken
342/13/91 A	Sally Baker, Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Udo Graf, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Rainer Knödlseder, Jaroslav Kremenek, Siegfried Langenbuch, Robert Lindhof, Thomas Ludwig, Peter Luksch, Roy Milner, Bernhard Ries, Thomas Treml: TOPSYS - Tools for Parallel Systems (Artikelsammlung); 2., erweiterte Auflage
342/14/91 A	Michael Griebel: The combination technique for the sparse grid solution of PDE's on multiprocessor machines
342/15/91 A	Thomas F. Gritzner, Manfred Broy: A Link Between Process Algebras and Abstract Relation Algebras?
342/16/91 A	Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, Roland Wismüller: The Design and Implementation of TOPSYS
342/17/91 A	Ulrich Furbach: Answers for disjunctive logic programs
342/18/91 A	Ulrich Furbach: Splitting as a source of parallelism in disjunctive logic programs
342/19/91 A	Gerhard W. Zumbusch: Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme
342/20/91 A	M. Jobmann, J. Schumann: Modelling and Performance Analysis of a Parallel Theorem Prover
342/21/91 A	Hans-Joachim Bungartz: An Adaptive Poisson Solver Using Hierarchical Bases and Sparse Grids
342/22/91 A	Wolfgang Ertel, Theodor Gemenis, Johann M. Ph. Schumann, Christian B. Suttner, Rainer Weber, Zongyan Qiu: Formalisms and Languages for Specifying Parallel Inference Systems
342/23/91 A	Astrid Kiehn: Local and Global Causes
342/24/91 A	Johann M.Ph. Schumann: Parallelization of Inference Systems by using an Abstract Machine
342/25/91 A	Eike Jessen: Speedup Analysis by Hierarchical Load Decomposition
342/26/91 A	Thomas F. Gritzner: A Simple Toy Example of a Distributed System: On the Design of a Connecting Switch
342/27/91 A	Thomas Schnekenburger, Andreas Weininger, Michael Friedrich: Introduction to the Parallel and Distributed Programming Language ParMod-C
342/28/91 A	Claus Dendorfer: Funktionale Modellierung eines Postsystems
342/29/91 A	Michael Griebel: Multilevel algorithms considered as iterative methods on indefinite systems

342/30/91 A	W. Reisig: Parallel Composition of Liveness
342/31/91 A	Thomas Bemmerl, Christian Kasperbauer, Martin Mairandres, Bernhard Ries: Programming Tools for Distributed Multiprocessor Computing Environments
342/32/91 A	Frank Leßke: On constructive specifications of abstract data types using temporal logic
342/1/92 A	L. Kanal, C.B. Suttner (Editors): Informal Proceedings of the Workshop on Parallel Processing for AI
342/2/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS
342/2-2/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS - Revised Version (erschienen im Januar 1993)
342/3/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems
342/4/92 A	Claus Dendorfer, Rainer Weber: Development and Implementation of a Communication Protocol - An Exercise in FOCUS
342/5/92 A	Michael Friedrich: Sprachmittel und Werkzeuge zur Unterstüt- zung paralleler und verteilter Programmierung
342/6/92 A	Thomas F. Gritzner: The Action Graph Model as a Link between Abstract Relation Algebras and Process-Algebraic Specifications
342/7/92 A	Sergei Gorlatch: Parallel Program Development for a Recursive Numerical Algorithm: a Case Study
342/8/92 A	Henning Spruth, Georg Sigl, Frank Johannes: Parallel Algorithms for Slicing Based Final Placement
342/9/92 A	Herbert Bauer, Christian Sporrer, Thomas Krodel: On Distributed Logic Simulation Using Time Warp
342/10/92 A	H. Bungartz, M. Griebel, U. Rüde: Extrapolation, Combination and Sparse Grid Techniques for Elliptic Boundary Value Problems
342/11/92 A	M. Griebel, W. Huber, U. Rüde, T. Störtkuhl: The Combination Technique for Parallel Sparse-Grid-Preconditioning and -Solution of PDEs on Multiprocessor Machines and Workstation Networks
342/12/92 A	Rolf Niedermeier, Peter Rossmanith: Optimal Parallel Algorithms for Computing Recursively Defined Functions
342/13/92 A	Rainer Weber: Eine Methodik für die formale Anforderungsspezifkation verteilter Systeme
342/14/92 A	Michael Griebel: Grid– and point–oriented multilevel algorithms
342/15/92 A	M. Griebel, C. Zenger, S. Zimmer: Improved multilevel algorithms for full and sparse grid problems
342/16/92 A	J. Desel, D. Gomm, E. Kindler, B. Paech, R. Walter: Bausteine eines kompositionalen Beweiskalküls für netzmodellierte Systeme

Reihe A

342/17/92 A	Frank Dederichs: Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen
342/18/92 A	Andreas Listl, Markus Pawlowski: Parallel Cache Management of a RDBMS
342/19/92 A	Erwin Loibl, Markus Pawlowski, Christian Roth: PART: A Parallel Relational Toolbox as Basis for the Optimization and Interpretation of Parallel Queries
342/20/92 A	Jörg Desel, Wolfgang Reisig: The Synthesis Problem of Petri Nets
342/21/92 A	Robert Balder, Christoph Zenger: The d-dimensional Helmholtz equation on sparse Grids
342/22/92 A	Ilko Michler: Neuronale Netzwerk-Paradigmen zum Erlernen von Heuristiken
342/23/92 A	Wolfgang Reisig: Elements of a Temporal Logic. Coping with Concurrency
342/24/92 A	T. Störtkuhl, Chr. Zenger, S. Zimmer: An asymptotic solution for the singularity at the angular point of the lid driven cavity
342/25/92 A	Ekkart Kindler: Invariants, Compositionality and Substitution
342/26/92 A	Thomas Bonk, Ulrich Rüde: Performance Analysis and Optimization of Numerically Intensive Programs
342/1/93 A	M. Griebel, V. Thurner: The Efficient Solution of Fluid Dynamics Problems by the Combination Technique
342/2/93 A	Ketil Stølen, Frank Dederichs, Rainer Weber: Assumption / Commitment Rules for Networks of Asynchronously Communicating Agents
342/3/93 A	Thomas Schnekenburger: A Definition of Efficiency of Parallel Programs in Multi-Tasking Environments
342/4/93 A	Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: A Proof of Convergence for the Combination Technique for the Laplace Equation Using Tools of Symbolic Computation
342/5/93 A	Manfred Kunde, Rolf Niedermeier, Peter Rossmanith: Faster Sorting and Routing on Grids with Diagonals
342/6/93 A	Michael Griebel, Peter Oswald: Remarks on the Abstract Theory of Additive and Multiplicative Schwarz Algorithms
342/7/93 A	Christian Sporrer, Herbert Bauer: Corolla Partitioning for Distributed Logic Simulation of VLSI Circuits
342/8/93 A	Herbert Bauer, Christian Sporrer: Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving
342/9/93 A	Peter Slavkovsky: The Visibility Problem for Single-Valued Surface (z = f(x,y)): The Analysis and the Parallelization of Algorithms
342/10/93 A	Ulrich Rüde: Multilevel, Extrapolation, and Sparse Grid Methods
342/11/93 A	Hans Regler, Ulrich Rüde: Layout Optimization with Algebraic Multigrid Methods
342/12/93 A	Dieter Barnard, Angelika Mader: Model Checking for the Modal Mu-Calculus using Gauß Elimination

Reihe A

342/13/93 A	Christoph Pflaum, Ulrich Rüde: Gauß' Adaptive Relaxation for the Multilevel Solution of Partial Differential Equations on Sparse Grids
342/14/93 A	Christoph Pflaum: Convergence of the Combination Technique for the Finite Element Solution of Poisson's Equation
342/15/93 A	Michael Luby, Wolfgang Ertel: Optimal Parallelization of Las Vegas Algorithms
342/16/93 A	Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: Pointwise Convergence of the Combination Technique for Laplace's Equation
342/17/93 A	Georg Stellner, Matthias Schumann, Stefan Lamberts, Thomas Ludwig, Arndt Bode, Martin Kiehl und Rainer Mehlhorn: Developing Multicomputer Applications on Networks of Workstations Using NXLib
342/18/93 A	Max Fuchs, Ketil Stølen: Development of a Distributed Min/Max Component
342/19/93 A	Johann K. Obermaier: Recovery and Transaction Management in Write-optimized Database Systems
342/20/93 A	Sergej Gorlatch: Deriving Efficient Parallel Programs by Systemating Coarsing Specification Parallelism
342/01/94 A	Reiner Hüttl, Michael Schneider: Parallel Adaptive Numerical Simulation
342/02/94 A	Henning Spruth, Frank Johannes: Parallel Routing of VLSI Circuits Based on Net Independency
342/03/94 A	Henning Spruth, Frank Johannes, Kurt Antreich: PHIroute: A Parallel Hierarchical Sea-of-Gates Router
342/04/94 A	Martin Kiehl, Rainer Mehlhorn, Matthias Schumann: Parallel Multiple Shooting for Optimal Control Problems Under NX/2
342/05/94 A	Christian Suttner, Christoph Goller, Peter Krauss, Klaus-Jörn Lange, Ludwig Thomas, Thomas Schnekenburger: Heuristic Optimization of Parallel Computations
342/06/94 A	Andreas Listl: Using Subpages for Cache Coherency Control in Parallel Database Systems
342/07/94 A	Manfred Broy, Ketil Stølen: Specification and Refinement of Finite Dataflow Networks - a Relational Approach
342/08/94 A	Katharina Spies: Funktionale Spezifikation eines Kommunikationsprotokolls
342/09/94 A	Peter A. Krauss: Applying a New Search Space Partitioning Method to Parallel Test Generation for Sequential Circuits
342/10/94 A	Manfred Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style
342/11/94 A	Eckhardt Holz, Ketil Stølen: An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus
342/12/94 A	Christoph Pflaum: A Multi-Level-Algorithm for the Finite-Element-Solution of General Second Order Elliptic Differential Equations on Adaptive Sparse Grids

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox - Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
	Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS