Transparent Data Structures, or How to Make Search Trees Robust in a Distributed Environment

Miroslaw Korzeniowski[*]
International Graduate School of
Dynamic Intelligent Systems
Computer Science Department
University of Paderborn
D-33102 Paderborn, Germany
rudy@upb.de

Christian Scheideler[†]
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
scheideler@cs.jhu.edu

Abstract

In this paper we propose a new class of memory models, called transparent memory models*, for implementing data structures so that they can be emulated in a distributed environment in a scalable, efficient and robust way. Transparent memory models aim at combining the advantages of the pointer model and the linear addressable memory model without inheriting their disadvantages. We demonstrate the effectiveness of our approach by looking at a specific memory model, called the* hypertree memory model*, and by implementing a search tree in it that matches, in an amortized sense, the performance of the best search trees in the pointer model yet can efficiently recover from* arbitrary *memory faults.*

1. Introduction

In this paper we propose a new class of memory models for implementing data structures so that they can be emulated in a distributed environment in a scalable, efficient, and robust way. *Scalability* means that a low degree overlay network can be used for the emulation of the data structure, *efficiency* means that accesses in the data structure can be emulated with a low time and work overhead, and *robustness* means that the data structure can recover from (a potentially large number of arbitrary or random) memory faults. Scalability, efficiency and robustness are especially important for dynamic distributed systems such as peer-to-peer systems which have recently attracted a lot of attention.

Most of the data structures in the algorithms literature are based on one of the following two basic models: the pointer model or the linear addressable memory model. In the linear addressable memory model we have a linear addressable memory and every read and write request to a memory cell can be processed at unit cost. In the pointer model we only have memory cells and labeled pointers interconnecting the memory cells. There is nothing like an addressable space. All read and write accesses have to be handled via pointers. The standard assumption is that forwarding a request along a pointer can be done at unit cost. For papers on various pointer machine models see, for example, [6, 9, 13, 25].

Both models are useful for the design of data structures for a single machine. However, in a distributed environment both models have their limitations. The basic problem with these models is that they give too much freedom in designing a data structure. This can make it tempting to design data structures that are hard to emulate in a distributed environment in a scalable, efficient, and robust way. We first discuss why this is so, and then we propose a new class of memory models called *transparent memory models*.

1.1. Linear addressable memory model

First, consider the problem of emulating a data structure in the linear addressable memory model in a distributed environment. Since in this model it may be hard to predict the access pattern without knowing the input in advance, a universal shared memory implementation usually has to be used. A vast number of shared memory platforms have already been devel-

* Partially supported by DFG-Sonderforschungsbereich 376 “Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen” and by the Future and Emerging Technologies programme of EU under EU Contract 001907 DELIS ”Dynamically Evolving, Large Scale Information Systems”.

† Supported by NSF grants CCR-0311121 and CCR-0311795.

oped. See `http://www.computer.org/parascope/` or `http://www.aspenleaf.com/distributed/` for a comprehensive list. Also several scalable systems that provide a shared memory platform in a peer-to-peer environment have been developed. The most prominent are Chord [24], CAN [22], Pastry [23], and Tapestry [16]. As with all of the existing systems, the problem with any scalable universal shared memory implementation is that it has an inherent (time and work) overhead of essentially $\Omega(\log n)$. (This is because by definition, a scalable solution requires the shared memory to be distributed in a system interconnected by an overlay network of at most polylogarithmic degree.) The time overhead may be reduced with the help of pipelining techniques but reducing the work overhead can be very difficult. It usually requires knowledge about the access structure so that parts of the data structure that have a high access correlation (such as a subheap in a heap) can be stored together or at a small hop-distance in the distributed system and therefore fewer transmissions of requests over the network are necessary. However, since in a pure shared memory model there is no incentive for a programmer to implement data structures that have a high degree of locality, there may not be a high access correlation that can be exploited.

Robustness can also be a problem. Shared memory platforms generally aim at providing a robust memory to the user. A general technique for this is to use redundancy. However, redundancy is expensive and cannot protect against all possible bad events that may happen. Therefore, it is important to consider also events in which an item in a data structure may not be recoverable.

1.2. Pointer model

If we use the pointer model instead of the linear addressable memory model, then we can get rid of some of the problems, such as identifying the access structure. For example, by maintaining a network connection for each pointer in the data structure, we can obtain a faithful (i.e., constant overhead) emulation of the original data structure in the distributed system. However, if the data structure is much larger than the number of processing units available for its emulation, then even though the data structure itself may be scalable, embedding it in the wrong way into the distributed system may result in a non-scalable overlay network for the emulation of the data structure. Embedding a data structure in the right way (i.e., with a low number of network connections for the non-local pointers) is a nontrivial problem in general and has been extensively investigated in the context of network embeddings and partitions in the past (e.g., [18, 19, 20]). This problem can be removed when embedding the data structure into a shared space, but as mentioned above, a scalable implementation of a shared space has an inherent overhead of $\Omega(\log n)$.

Data structures based on the pointer model can have serious problems with robustness if they are not designed carefully. Pointer structures such as linear lists and trees, for example, are not useful for a distributed environment because a single failure of a memory cell (such as the root of a search tree) can make the whole data structure or a large part of it inaccessible. Therefore, also more fault-tolerant pointer structures have been investigated. See, for instance, the work by Aumann and Bender [4] or on skip graphs [2], skip nets [14], and deterministic variants of the skip graph [5, 15]. Since these pointer structures have a high expansion, they can suffer many memory cell faults and still have a large connected component. However, those parts disconnected from the rest of the structure are lost. Our goal, instead, is to find mechanisms so that *all* information in the data structure that did not get lost due to memory faults can be recovered and reorganized so that the data structure is back to a legal state.

1.3. Our contributions

Our contributions are threefold:

- We introduce a class of structured memory models that we call *transparent memory models*. Transparent memory models are memory models in which memory accesses can be emulated in a scalable overlay network with constant work.
- As a specific example, we introduce a transparent memory model called the *hypertree memory model* and design a scalable, dynamic overlay network that can emulate memory accesses in this model with constant work.
- We show how to implement a search tree in the hypertree memory model with the property that the amortized work for insert, delete and search is the same as for the best search trees in the pointer model, yet it can efficiently recover *all* remaining information under *arbitrary* memory faults. Moreover, in the emulation, every node in the dynamic overlay network only has to perform a worst-case logarithmic amount of work for any insert, delete or search operation.

Next we talk about these contributions in more detail.

1.4. Transparent memory models

Transparent memory models belong to the class of structured memory models. In a structured memory model there is a countably infinite set U of memory cells and an infinite family of pointer structures $\mathcal{H} = \{H_n = (U_n, F_n) \mid n \in$

$\mathbb{N}, U_n \subseteq U, F_n \subseteq U_n \times U_n\}$ interconnecting these memory cells so that $U_n \subseteq U_{n+1}$, $F_n \subseteq F_{n+1}$, and H_n is connected for all $n \in \mathbb{N}$. Requests can only be exchanged between adjacent cells and every such request can be processed at unit cost. We call a structured memory model *transparent* if there is an infinite family $\mathcal{G} = \{G_n = (V_n, E_n) \mid n \in \mathbb{N}\}$ of graphs of at most polylogarithmic degree so that for all $n \in \mathbb{N}$ and $m \in \mathbb{N}$, H_m can be mapped to G_n with dilation at most 1 and load at most $\tilde{O}(|U_m|/|V_n|)$ (where $\tilde{O}$ suppresses logarithmic factors). That is, for all n and m, there is a mapping $f : U_m \to V_n$ of the cells in H_m to the nodes in G_n so that for all $v \in V_n$, $|f^{-1}(v)| = \tilde{O}(|U_m|/|V_n|)$, and for all $\{v, w\} \in F_m$, $f(v) = f(w)$ or $\{f(v), f(w)\} \in E_n$.

In order to demonstrate the usefulness of transparent memory models, we focus on a specific model called the hypertree memory model and implement an efficient and robust search tree in it. We also present a scalable overlay network design that can emulate the hypertree memory model with dilation 1. The idea behind suggesting transparent memory models goes back to a result by Naor and Wieder [21] demonstrating how to embed replication trees with dilation 1 in a scalable overlay network in order to relieve hot spots in a peer-to-peer system.

1.5. Related work on search structures

Work on search structures has a long history. The most popular search structures are probably AVL trees, red-black trees, skip lists, and splay trees. Whereas all of these structure allow to process insert, delete, and search operations with a cost of $O(\log n)$ and are therefore efficient, none of these structures is robust to memory faults.

There are basically two approaches of making search structures robust to memory faults: using high-expansion pointer structures such as the skip graph [2] or hyperring [5], or embedding the search structure in a compact form into an array [1, 7, 8, 17]. The first approach cannot recover from arbitrary memory faults but can at least make sure that the number of non-faulty entries that get disconnected from the largest connected component of non-faulty entries is within a constant factor of the faulty entries [3]. The second approach allows, in principle, recovery from arbitrary memory faults, but it is not clear how much time would be needed for that. But even if there were an efficient recovery mechanism, the fact that insert and delete operations in these search structures require an amortized work of $\Theta(\log^2 n)$ [1, 7, 8, 17], which is also believed to be best possible [7, 12, 11], makes them not particularly attractive.

Hence, it is rather surprising that we can demonstrate the feasibility of a search structure with amortized $O(\log n)$ work for insert and delete operations and worst case $O(\log n)$ work for search operations that can recover efficiently to a maximum possible extent from *arbitrary* memory faults.

1.6. Structure of the paper

Section 2 presents the hypertree memory model and demonstrates that it can be emulated in a scalable, efficient, and robust way, and Section 3 shows how to implement an efficient and robust search tree in the hypertree model. Due to space constraints, most of the proofs are left out.

2. Hypertree Memory Model

In the hypertree memory model, $U = \{0, 1\}^*$, i.e., the cells in the model are labeled by binary strings. The class $\mathcal{H}$ of pointer structures interconnecting these cells is defined as follows. (Recall that the *Hamming distance* $H(v, w)$ of any two bit strings $v, w \in \{0, 1\}^k$ is equal to $\sum_{i=1}^{k} |v_i - w_i|$.)

Definition 2.1 *A* hypertree of depth *d*, $H_d = (U_d, F_d)$, *is a pointer structure on* $U_d = \{u \in U \mid |u| \leq d\}$ *in which*

- *the root has the label* ϵ *(the empty label),*
- *every node* $v \in U_{d-1}$ *is connected to* $v0$ *and* $v1$ *(the tree edges) and*
- *every node* $v \in U_d$ *is connected to every node* w *with* $|v| = |w|$ *and Hamming distance* $H(v, w) = 1$ *(the hypercube edges), i.e., v and w only differ in one bit.*

We designed the hypertree specifically to support an efficient implementation of a search tree because the tree edges will be needed for the search tree structure and the hypercube edges will be needed for the balancing. In the following, we demonstrate that the hypertree memory model can be emulated by an infinite graph family so that the transparent memory model conditions are satisfied.

2.1. A family of graphs for the hypertree model

Consider the following two well-known classes of graphs.

Definition 2.2 (de Bruijn) *For any* $d \geq 0$*, the* d-dimensional de Bruijn graph $DB(d)$ *is an undirected graph* $G = (V, E)$ *with node set* $V = \{0, 1\}^d$ *and edge set* E *that contains all edges* $\{v, w\}$ *with* $v = (v_1, \ldots, v_d)$ *and* $w \in \{(x, v_1, \ldots, v_{d-1}) \mid x \in \{0, 1\}\}$.

Definition 2.3 (Hypercube) *For any* $d \geq 0$*, the* d-dimensional hypercube $HC(d)$ *is an undirected graph* $G' = (V, E')$ *with node set* $V = \{0, 1\}^d$ *and edge set* E' *that contains an edge between any two nodes* $v, w \in V$ *with* $H(v, w) = 1$.

When combining these two graphs, we obtain the following new class of graphs.

Definition 2.4 (de Bruijn cube) *For any $d \geq 0$, the d-dimensional* de Bruijn cube $DC(d)$ *is an undirected graph $G_d = (V_d, E_d)$ with node set $V_d = \{0,1\}^d$ and edge set $E_d = E \cup E'$.*

Given a finite binary string $b = (b_1 b_2 \ldots b_k)$, let $b^R = (b_k \ldots b_2 b_1)$ and $\mathrm{prefix}_{k'}(b) = (b_1 \ldots b_{\min\{k,k'\}})$. For any two binary strings a and b, $a \circ b$ represents their concatenation. In order to map the cells of any H_d to the nodes of any $G_{d'}$, we use the following mapping $f_{d,d'} : U_d \to V_{d'}$:

$$f_{d,d'}(s) = \mathrm{prefix}_{d'}(s^R \circ r) \quad \text{for all } s \in U_d \setminus \{\epsilon\}$$

where $r \in \{0,1\}^{d'}$ is the bit string the root ϵ is mapped to in f and can be selected in an arbitrary way. This mapping has the following important property, which is easy to show:

Theorem 2.5 *For any $d \in \mathbb{N}$, $d' \in \mathbb{N}$ and $r \in \{0,1\}^{d'}$, the embedding of H_d into $G_{d'}$ via $f_{d,d'}$ has a dilation of at most 1 and a load of at most $|U_d|/|V_{d'}| + d'$.*

Hence, the hypertree model satisfies the properties of a transparent memory model.

2.2. Dynamic graphs for the hypertree model

Using the continuous-discrete approach of Naor and Wieder [21], one can transform the family of de Bruijn cubes into a dynamic de Bruijn cube suitable for peer-to-peer systems. Interpreting every binary label $(v_1, \ldots, v_d)$ as a point $x = \sum_{i=0}^{d-1} v_i/2^i$ in $[0,1)$, we can formulate the following continuous variant of the de Bruijn cube.

Definition 2.6 *The* continuous de Bruijn cube *consists of the space $V = [0,1)$ and a set of functions $f_0, f_1 : V \to V$ and $g_i : V \to V$, $i \geq 1$ with*

- *$f_i(x) = (x+i)/2$ for all $i \in \{0,1\}$ (which represents the de Bruijn edges) and*
- *$g_i(x) = x \oplus 2^{-i}$ for all $i \in \mathbb{N}$ (which represents the hypercube edges), where $x \oplus y$ is the bit-wise XOR of (the binary representations of) x and y.*

This continuous form can then be converted back into a discrete form following [21] that allows cheap update costs if nodes join or leave the graph so that, for n nodes, any hypertree space H_d can still be mapped with dilation at most 1 and load $O((|U_d|/n)\log n)$, with high probability.

2.3. Robustness

The hypertree space model has the advantage that, in principle, it allows efficient recovery from *arbitrary* memory faults (as long as the distributed system emulating it can recover). We call a data structure implementation in the hypertree model *compact* if, in the fault-free state, the memory cells of the data structure form a connected component in the hypertree. A data structure is called *recoverable* if it has a recovery mechanism allowing it to recover from arbitrary lost memory cells. Suppose that we use the following strategy whenever a node v establishes a new edge to a node w in the de Bruijn cube: *Wake up the recovery mechanism in all used memory cells assigned to v.* Then the following result holds:

Theorem 2.7 *Any compact data structure implementation in the hypertree model that can recover from arbitrary lost memory cells can be efficiently emulated by a scalable overlay network so that also in the emulation it can recover from arbitrary lost memory cells.*

Proof. (Sketch) Follows from the fact that the data structure is compact and that any lost memory cell will eventually be detected once the edge in the network emulating a connection to a still working memory cell has been recovered. □

We use the term "efficient" in the theorem because the recovery mechanism only needs to be invoked in the case that there is an edge change in the network. As long as the network is static, no checks have to be performed (under the assumption that local memory is reliable). Hence, Section 2 implies that efficiency and recovery properties in the abstract hypertree model are transferrable to its emulation.

3. Search Tree

In this section we show how to implement a balanced search tree in the hypertree model. We use amortized analysis to show that our construction has low average cost per operation. We define the tree in the following way.

Condition 3.1 *For each node r of the tree T, where the subtrees rooted in the left and right child of r are α and β, respectively:*

1. *r stores at most one entry*
2. *if r stores an entry and r is not the root, also the parent of r stores an entry*
3. *for all $a \in \alpha$ and $b \in \beta$, $a \leq r \leq b$ (i.e., the tree is sorted)*
4. *$\frac{|\alpha|}{3} - 1 \leq |\beta| \leq 3 \cdot |\alpha| + 1$ (i.e., the tree is balanced)*
5. *r stores the weight (i.e., size) of the subtree rooted in it*

In order to perform operations on the tree and maintain the balance in it we use the following primitives, each of them working on a subtree T_v rooted in a node v:

move upwards v is moved to its parent and T_v is moved so that it remains the subtree rooted in v

move downwards v is moved to its child and T_v is moved so that it remains the subtree rooted in v

move sideways v is moved to its tree-sibling and T_v is moved so that it remains the subtree rooted in v

The following lemma states the time and communication cost of all primitive procedures:

Lemma 3.2 *Each of the primitives*

- *takes logarithmic time in the size of the moved subtree*
- *needs communication work linear in the size of the moved subtree*

The basic tree operations are performed as follows. Each of them starts in the root of the tree.

search(a) If the element in the root is equal to a, the address of the root is returned. Otherwise the search is performed recursively in the left (if a is smaller than the value in the root) or right (if a is larger) subtree. When the subtree does not exist (e.g. when we are in the leaf), the element is reported to be not found.

insert(a) First, search(a) is performed, and if a is found, the operation terminates. If a is not found, then a is inserted at the proper place below the leaf reached by search(a).

delete(a) First, search(a) is performed. Let x be the found node (if it exists). If it is a leaf, it is simply removed from the tree. If not, let y be the rightmost node in its left subtree (or the leftmost node in its right subtree if the left subtree is empty). We remove a from x and move the element from y to x. The node y was either a leaf or its right (left) subtree was empty. If it was a leaf, we can simply remove it. Otherwise the whole non-empty subtree is moved upwards.

We show that as long as the tree is balanced, each of the above operations takes $O(\log n)$ time, where n is the current number of nodes.

Lemma 3.3 *An isolated search or insert operation or any m consecutive delete operations preserve Condition 3.1 (except point 4) in the tree with logarithmic cost per operation.*

After performing an operation on the tree it is easy to check whether the tree is still balanced as each node that participated in the operation can check whether its children fulfill the balance condition (Condition 3.1, property 4). If they do not, a rebalancing routine is needed. We give a routine which has logarithmic amortized cost per operation and performs any rebalancing in worst-case logarithmic time. The latter implies that each node participating in the network has to send $O(\log n)$ messages during the rebalancing routine, i. e. the approach is fair.

3.1. Amortized analysis

In order to analyze the cost of the algorithm we use amortized analysis. Instead of defining the potential for the tree as a whole, we define it for each node separately.

Definition 3.4 *For a tree T with subtrees α and β, the* potential *stored in its root r is the difference between the weights of α and β: $\phi_r = ||\alpha| - |\beta||$*

Obviously, for each node r the potential in r is never negative. Thus, the potential of the whole tree T, which is the sum of the potentials of all nodes, is never negative either. We prove that if we can attribute $\Theta(\log n)$ additional cost (in the form of virtual coins) to each insert and delete operation, we can use this capital to ensure that the potential in all nodes of the tree is in accordance with the definition. Later, we can use the stored coins to pay for a rebalancing routine.

Lemma 3.5 *For an insert or delete operation, $O(\log n)$ additional coins suffice to update the potentials in the tree where it is necessary.*

Our algorithm uses rotations, similar to other approaches (see for example [10], the chapter about red-black trees), in order to keep the tree balanced. We can show the following properties.

Lemma 3.6 *If for a node r, the subtrees rooted in the children of r are balanced (all nodes fulfill Condition 3.1(4), called the balance condition further on), then local balancing via move upwards/downwards/sideways operations can modify the tree so that*

- *r fulfills the balance condition*
- *on each path from r to a leaf at most one node does not fulfill the balance condition*
- *only a constant number of move operations is used*
- *the potential stored in the tree decreases and the total communication cost is at most by a constant factor larger than the decrease in potential*

For a node r the local balancing procedure checks 3 levels of descendants of r and depending on the weights of the trees rooted in them it performs certain rotations (not explained here). If some other balancing procedures are currently being executed lower in the tree they do not influence the procedure in r. It is only important that no node from the highest 3 levels is participating in any rotations at the same time. The trees below the third level are only moved as a whole and individual nodes of such trees are moved after they have finished the rotations started earlier.

The balancing of the tree is done in a bottom-up-down fashion. The information about imbalance comes from below and a node r performs a balance procedure on itself.

The latter can destroy the balance in the children of r but thanks to Lemma 3.6, the children can run the new balancing procedures without delay. Thus, after a constant number of steps, the information about imbalance can be forwarded to the parent of r.

Theorem 3.7 *The total communication cost of the balancing procedure is $O(\Delta_\phi + m \cdot \log n)$ and the time of the balancing procedure is $O(\log n)$, where Δ_ϕ is the total decrease of potential, n is the initial number of nodes in the tree and m is the number of insert or delete operations executed right before the balancing.*

3.2. Robustness

The search tree can also recover fast from arbitrary memory faults. Before a failure, the tree is balanced and each node knows its height in it. After the failure this information is used to guess the maximum depth of survivors, and the surviving nodes wait for information from them. The latter takes time proportional to the original depth of the tree. The communication cost is at most a constant number of messages per failure, as only the children of broken nodes initiate the repair procedure. After gathering the information about broken nodes, the tree can be compressed as though the broken nodes were deleted in a normal way.

References

[1] A. Andersson and T.W. Lai. Fast updating of well-balanced trees. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 111–121, 1990.

[2] J. Aspnes and G. Shah. Skip graphs. In *Proc. of the 14th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 384–393, 2003.

[3] J. Aspnes and U. Wieder. The expansion and mixing time of skip graphs with applications. In *Proc. of the 17th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 2005.

[4] Y. Aumann and M.A. Bender. Fault tolerant data structures. In *Proc. of the 37th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 580–589, 1996.

[5] B. Awerbuch and C. Scheideler. The hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. of the 15th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 2004.

[6] A.M. Ben-Amram and Z. Galil. On pointers versus addresses. *Journal of the ACM*, 39(3):617–648, 1992.

[7] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. of the 41st IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[8] G. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via trees of small height. In *Proc. of the 13th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[9] S.A. Cook and P.W.Dymond. Parallel pointer machines. *Computational Complexity*, 3:19–30, 1993.

[10] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1999.

[11] P.F. Dietz, J.I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proc. of the 6th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 131–142, 1994.

[12] P.F. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–180, 1990.

[13] M.T. Goodrich and S.R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *Proc. of the 30th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 190–195, 1989.

[14] N. J. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[15] N.J. Harvey and I. Munro. Brief announcement: Deterministic skipnet. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing (PODC)*, 2003.

[16] K. Hildrum, J.D. Kubiatowicz, S. Rao, and B.Y. Zhao. Distributed object location in a dynamic network. In *Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 41–52, 2002.

[17] A. Itai, A.G. Konheim, and M. Rodeh. A sparse table implementation of sorted sets. Technical Report Research Report RC 9146, IBM T.J. Watson Research Center, Yorktown Heights, New York, November 1981.

[18] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, 1992.

[19] B. Monien. Software packages for graph partitioning. Dept. of Computer Science, Unversity of Paderborn, see http://wwwcs.uni-paderborn.de/fachbereich/AG/monien /SOFTWARE.

[20] B. Monien and H. Sudborough. Embedding one interconnection network in another. *Computing Suppl.*, 7:257–282, 1990.

[21] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proc. of the 15th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 50–59, 2003.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of the ACM SIGCOMM*, 2001.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM*, pages 149–160, 2001.

[25] P. van Emde Boas. *Handbook of Theoretical Computer Science, Vol. A*, chapter Machine models and simulations, pages 1–66. Elsevier, 1990.