Deterministic load balancing and dictionaries in the parallel disk model

Mette Berger* Esben Rune Hansen† Rasmus Pagh†

Mihai Pătraşcu‡ Milan Ružić† Peter Tiedemann†

ABSTRACT

We consider deterministic dictionaries in the parallel disk model, motivated by applications such as file systems. Our main results show that if the number of disks is moderately large (at least logarithmic in the size of the universe from which keys come), performance similar to the expected performance of randomized dictionaries can be achieved. Thus, we may avoid randomization by extending parallelism. We give several algorithms with different performance tradeoffs. One of our main tools is a deterministic load balancing scheme based on expander graphs, that may be of independent interest.

Our algorithms assume access to certain expander graphs "for free". While current explicit constructions of expander graphs have suboptimal parameters, we show how to get near-optimal expanders for the case where the amount of data is polynomially related to the size of internal memory.

Categories and Subject Descriptors

E.2 [**Data**]: Data Storage Representations; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Performance, Theory

Keywords

Deterministic, dictionary, parallel disk model, expander graph, hashing

*OctoShape ApS

†IT University of Copenhagen, Rued Langgaards Vej 7, 2300 København S, Denmark.

‡MIT

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA'06, July 30–August 2, 2006, Cambridge, Massachusetts, USA.

Copyright 2006 ACM 1-59593-452-9/06/0007 ...$5.00.

1. INTRODUCTION

Storage systems for large data sets are increasingly *parallel*. There are now disk arrays consisting of more than a thousand disks[1], and *Network-Attached Storage* (NAS) solutions could in principle scale up to an arbitrary number of storage servers. A simple, feasible model for these kinds of situations is the parallel disk model [19]. In this model there are D storage devices, each consisting of an array of memory blocks with capacity for B *data items*; a data item is assumed to be sufficiently large to hold a pointer value or a *key* value. The performance of an algorithm is measured in the number of parallel I/Os, where one parallel I/O consists of retrieving (or writing) a block of B data items from (or to) each of the D storage devices.

The problem studied in this paper is the design of *dictionaries*, i.e., data structures storing a set of n keys from some bounded universe U, as well as "satellite" information associated with each key, supporting lookups of keys and dynamic updates to the key set. This is a fundamental and well-studied problem in computer science. In the context of external memory, note that a dictionary can be used to implement the basic functionality of a file system: Let keys consist of a file name and a block number, and associate them with the contents of the given block number of the given file. Note that this implementation gives "random access" to any position in a file.

Most external memory algorithms for one disk can improve their performance by a factor of D in the parallel disk model using *striping*. For external memory data structures we can expect no such improvement, since at least one I/O is needed to answer on-line queries. For example, the query time of a B-tree in the parallel disk model is $\Theta(\log_{BD} n)$, which means that no asymptotic speedup is achieved compared to the one disk case unless the number of disks is very large, $D = B^{\omega(1)}$. Randomized dictionaries based on hashing support lookups and updates in close to 1 I/O per operation, as in the single disk case.

1.1 Our results and comparison with hashing

In this paper we show a new kind of benefit by moving from one to many disks: Efficient randomized dictionaries may be replaced by deterministic ones of essentially the same efficiency. Besides the practical problem of giving an algorithm access to random bits, randomized solutions never give firm guarantees on performance. In particular, all hashing based dictionaries we are aware of may use $n/B^{O(1)}$ I/Os

[1]For example, the Hitachi TagmaStore USP1100 disk array can include up to 1152 disks, storing up to 32 petabytes.

for a single operation in the worst case. In contrast, we give very good guarantees on the worst case performance of any operation. No previously known dynamic dictionary in a feasible model of computation has constant worst-case cost for all operations and linear space usage.

Randomized dictionaries

The most efficient randomized dictionaries, both in theory and practice, are based on hashing techniques. Much work has been devoted to the trade-off between time and space for such dictionaries, but in the context of this paper we will only require that a dictionary uses linear space. While hashing algorithms were historically analyzed under the so-called *uniform hashing* assumption, most modern results are shown using explicit, and efficiently implementable, hash functions. In the context of external memory, the key requirement on a hash function is that its description should fit into internal memory. A reasonable assumption, made in the following, is that internal memory has capacity to hold $O(\log n)$ keys. This allows $O(\log n)$-wise independent hash functions, for which a large range of hashing algorithms can be shown to work well, see e.g. [14, 15].

There are dictionaries having performance that is, in an asymptotic sense, almost as good as one can hope for in a randomized structure. If one wants a randomized dictionary such that bounds on running times of its operations can be expressed in form $O(1)$, without interest in the actual constant, then a choice is the dictionary of [7], having lookup and update costs of $O(1)$ I/Os with high probability (the probability is $1 - O(n^{-c})$, where c can be chosen as any constant).

In our setting, having D parallel disks can be exploited by *striping*, i.e., considering the disks as a single disk with block size BD. If BD is at least logarithmic in the number of keys, a linear space hash table (with a suitable constant) has no overflowing blocks with high probability. This is true even if we store associated information of size $O(BD/\log n)$ along with each key. Note that one can always use the dictionary to retrieve a pointer to satellite information of size BD, which can then be retrieved in an extra I/O. However, it is interesting how much satellite information can be returned in a single I/O. We call the maximum supported size of satellite data of a given method its *bandwidth*. Cuckoo hashing [13] can be used to achieve bandwidth $BD/2$, using a single parallel I/O, but its update complexity is only constant in the amortized expected sense.

In general, the *average* cost of an operation can be made arbitrarily close to 1, whp., by the following folklore trick: Keep a hash table storing all keys that do not collide with another key (in that hash table), and mark all locations for which there is a collision. The remaining keys are stored using the algorithm of [7]. The fraction of searches and updates that need to go to the dictionary of [7] can be made arbitrarily small by choosing the hash table size with a suitably large constant on the linear term. Note that the bandwidth for this method can be made $\Theta(BD)$ by allowing extra I/Os for operations on the keys in the second dictionary.

Our results

In this paper we present *deterministic* and worst-case efficient results closely matching what can be achieved using hashing. All of our dictionaries use linear space. The first of our dictionaries achieves $O(1)$ I/Os in the worst case for both lookups and queries, without imposing any requirements on the value of B. A variation of this dictionary performs lookups in 1 I/O and updates in 2 I/Os, but requires that $B = \Omega(\log n)$; the bandwidth is $O(\frac{BD}{\log n})$, like in the comparable hashing result. We also give another dictionary that achieves the bandwidth of $O(BD)$ at the cost of relaxed operation performance – lookups take $1 + \epsilon$ I/Os on average, and updates take $2+\epsilon$ I/Os on average. The worst-case cost is $O(\log n)$, as opposed to hashing where the worst-case cost is usually linear. Figure 1.1 shows an overview of main characteristics of all the mentioned dictionaries – the new ones and the comparable hashing-based structures.

Method	**Lookup I/Os**	**Update I/Os**	**Bandwidth**	**Conditions**
[7]	$O(1)$	$O(1)$ whp.	-	-
Section 4.1	$O(1)$	$O(1)$	-	$D=\Omega(\log u)$
Hashing, no overflow	1 whp.	2 whp.	$O\left(\frac{BD}{\log n}\right)$	$BD=\Omega(\log n)$
Section 4.1	1	2	$O\left(\frac{BD}{\log n}\right)$	$D=\Omega(\log u)$ $B=\Omega(\log n)$
[13]	1	$O(1)$ am. exp.	$O(BD)$	-
[7] + trick	$1+\epsilon$ avg. whp.	$2+\epsilon$ avg. whp.	$O(BD)$	-
Section 4.3	$1+\epsilon$ avg.	$2+\epsilon$ avg.	$O(BD)$	$D=\Omega(\log u)$ $B=\Omega(\log n)$

Figure 1: Old and new results for linear space dictionaries with constant time per operation. The parameter ϵ can be chosen to be any positive constant; u denotes the size of the universe U. Update bounds take into account the cost of reading a block before it can be written, making 2 I/Os the best possible. Bounds that hold on average over all elements in the set are marked "avg." Where relevant (for dictionaries using close to 1 I/O), the bandwidth is also stated.

All of our algorithms share features that make them suitable for an environment with many concurrent lookups and updates:

- There is no notion of an index structure or central directory of keys. Lookups and updates go directly to the relevant blocks, without any knowledge of the current data other than the size of the data structure and the size of the universe U.
- If we fix the capacity of the data structure and there are no deletions (or if we do not require that space of deleted items is reused), no piece of data is ever moved, once inserted. This makes it easy to keep references to data, and also simplifies concurrency control mechanisms such as locking.

The results in Figure 1.1 are achieved by use of an unbalanced expander graph of degree $O(\log u)$. While the existence of such a graph is known, the currently best explicit (i.e., computationally efficient) construction has degree $2^{(\log\log u)^{O(1)}}$ [17]. This means that with current state-of-the-art in expander construction, the smallest number of disks for which we can realize our scheme is $2^{(\log\log u)^{O(1)}}$.

In Section 5 we explore the situation in which we have $\sqrt[b]{u}$ words of internal memory available, for some constant b, and obtain a "semi-explicit" expander construction of degree $(\log u)^{O(1)}$. The resulting expanders require either a factor $(\log u)^{O(1)}$ more space or a slightly stronger model (the *parallel disk head* model) to support our algorithms. The presented dictionary structures may become a practical choice if and when explicit and efficient constructions of unbalanced expander graphs appear.

1.2 Motivation

File systems are by excellence an associative memory. This associative retrieval is implemented in most commercial systems through variations of B-trees. In a UNIX example, to retrieve a random block from a file (inode), one follows pointers down a tree with branching factor B; leaves hold pointers to the blocks of the file.

Since one does not need the additional properties of B-trees (such as range searching), a hash table implementation can be better. In theory, this can save an $O(\log_B n)$ factor. In practice, this factor is a small constant: in most settings it takes 3 disk accesses before the contents of the block is available. However, the file system is of critical importance to overall performance, and making just one disk read instead of 3 can have a tremendous impact. Furthermore, using a hash table can eliminate the overhead of translating the file name into an inode (which we have not counted above), since the name can be easily hashed as well.

Note that the above justification applies only to random accesses, since for sequential scanning of large files, the overhead of B-trees is negligible (due to caching). One may question the need for such random access. For algorithms on massive data sets it is indeed not essential. However, there are also critical applications of a more data-structural flavor. Popular examples include webmail or http servers. These typically have to retrieve small quantities of information at a time, typically fitting within a block, but from a very large data set, in a highly random fashion (depending on the desires of an arbitrary set of users). Arrays of disks are of course the medium of choice for such systems, so parallelism is readily available. Our results show how parallelism can, at least in theory, be used to provide an attractive alternative to B-trees in such settings.

From a theoretical perspective, we observe a trade-off between randomness and parallelism that has not been explored before. But our main motivation comes from looking at the potential applications. Randomization at the file-system level is an idea that is often frowned upon. For one, having to deal with expected running times adds unnecessary complications to a critical component of the operating system, and a potential for malfunction. More importantly, the file system often needs to offer a real-time guarantee for the sake of applications, which essentially prohibits randomized solutions, as well as amortized bounds.

1.3 Related work

The idea of using expander graphs for dictionaries previously appeared in [5, 11, 4]. The results of [5] can be used to make a *static* dictionary, i.e., not supporting updates, in the parallel disk model, performing lookups in 1 I/O. The results of [11], which give a randomized structure for a serial RAM, can be modified to get a deterministic dynamic dictionary in a parallel setting [4]. That dictionary has good *amortized* bounds on the time for updates, but analyzed in the *parallel disk head model* [1] (one disk with D read/write heads), which is stronger than the parallel disk model, and fails to model existing hardware. Additionally, the worst-case cost of updates was shown in [4] to be linear in n. Our dictionaries have good worst-case performance, the I/O bounds on operations hold in the parallel disk model, and the methods are even simpler, in implementation as well as analysis, than the method of [11].

Other efforts towards efficient deterministic dictionaries (on a serial RAM) can be seen as derandomizations of hashing algorithms. However, the currently best methods need update time $n^{\Omega(1)}$ to achieve constant lookup time [10].

1.4 Overview of paper

In Section 2 we present definitions and notation to be used throughout the paper. One of our main tools, a deterministic load balancing scheme is presented in Section 3. In section 4.1 we explain how to use the load balancing scheme to get an efficient dictionary in the parallel disk model. Section 4.2 presents another way of using expanders to get an efficient dictionary in the parallel disk model, in the static case where there are no updates. In Section 4.3 this scheme is dynamized to get a scheme that uses close to the optimal number of I/Os for operations, on average over all elements. Finally, Section 5 presents a new explicit expander construction for external memory algorithms.

2. PRELIMINARIES

An essential tool, common to all of our dictionary constructions is a class of expander graphs. There have been a number of definitions of expander graphs, some of them equivalent, and different notations have been used in the literature. The graphs that we use are bipartite. In a bipartite graph $G = (U, V, E)$, we may refer to U as the "left" part, and refer to V as the "right" part; a vertex belonging to the left (right) part is called a left (right) vertex. In our dictionary constructions, the left part corresponds to the universe U of keys, and the right part corresponds to the disk blocks of the data structure. In Sect 5, where a new explicit construction is described, expanders are not tied to any particular application. A bipartite graph is called *left-d-regular* if every vertex in the left part has exactly d neighbors in the right part.

DEFINITION 1. *A bipartite, left-d-regular graph $G = (U, V, E)$ is a (d, ε, δ)-expander if any set $S \subset U$ has at least $\min((1-\varepsilon)d|S|,\ (1-\delta)|V|)$ neighbors.*

Since expander graphs are interesting only when $|V| < d|U|$, some vertices must share neighbors, and hence the parameter ε cannot be smaller than $1/d$.

We introduce notation for the cardinalities of important sets: $u = |U|$, $v = |V|$, and $n = |S|$. The set of neighbors of a set $S \subset U$ is denoted by

$$\Gamma_G(S) = \{y \in V \mid (\exists x \in S)\ (x, y) \in E\}\ .$$

The subscript G will be omitted when it is understood, and we write $\Gamma(x)$ as a shorthand for $\Gamma(\{x\})$. We use $[x]$ to denote the set $\{1, \ldots, x\}$. In Section 5 we will refer to bipartite expander graphs in terms of their neighbor function $F : U \times [d] \to V$, where for $x \in U$ $F(x, i)$ is the ith neighbor of x.

When G is an (d, ε, δ)-expander, then for any set $S \subset U$ such that $|S| < \frac{(1-\delta)v}{(1-\varepsilon)d}$ it holds that $|\Gamma(S)| \geq (1-\varepsilon)d|S|$. It will be convenient to introduce another *notational* definition of expander graphs; this definition is used starting from Section 4.2 and until the end of the paper.

DEFINITION 2. *A bipartite, left d-regular graph $G = (U, V, E)$ is an (N, ε)-expander if any set $S \subset U$ of at most N left vertices has at least $(1-\varepsilon)d|S|$ neighbors.*

There exist (d, ε, δ)-expanders with left degree $d = O(\log(\frac{u}{v}))$, for any v and positive constants ε, δ. If we wish that every subset of U having less than N elements expands "very well", i.e. if we need a (N, ε)-expander it is possible to have $v = \Theta(Nd)$ (clearly it is a requirement that $v = \Omega(Nd)$).

For applications one needs an *explicit* expander, i.e., an expander for which we can efficiently compute the neighbor set of a given node (in the left part). In the context of external memory algorithms, our requirement on an explicit construction is that the neighbor set can be computed without doing any I/Os, i.e., using only internal memory. No explicit constructions with the mentioned (optimal) parameters are known – see Section 5 for a discussion of state-of-the-art.

To make our algorithms work in the parallel disk model, we require an additional property of the expander graph: It should be *striped*. In a striped, d-regular, bipartite graph there is a partition of the right side into d sets such that any left vertex has exactly one neighbor in each set of the partition. The notion of an explicit construction for striped expander graphs has the additional requirement that $\Gamma(x)$ for any given x, should be returned in form (i, j), where i is the index of the partition set and j is the index within that set. The results from random constructions mentioned above hold even for striped random graphs. Unfortunately, most explicit constructions, in addition to not achieving optimal parameters, are also not striped.

3. DETERMINISTIC LOAD BALANCING

We will consider d-choice load balancing using (unbalanced) bipartite expander graphs. Suppose there is an unknown set of n left vertices where each vertex has k items, and each item must be assigned to one of the *neighboring* right vertices (called "buckets"). We consider a natural greedy strategy for balancing the number of items assigned to each bucket. The assumption is that the set is revealed element by element, and the decision on where to assign the k items must be made on-line. The strategy is this: Assign the k items of a vertex one by one, putting each item in a bucket that currently has the fewest items assigned, breaking ties arbitrarily. The scheme allows multiple items belonging to a vertex to be placed in one neighboring bucket.

A special case of this load balancing scheme, where $k = 1$ and the bipartite graph is a random graph of left degree 2, was presented and analyzed in [2, 3]. Tight bounds on the maximum load were given for the "heavily loaded case", showing that the deviation from the average load is $O(\log \log n)$ with a high probability. We now give an analogous result for a fixed (d, ε, δ)-expander. The scheme places a number of items in each bucket that is close to the average load of kn/v.

LEMMA 3. *If $d > \frac{k}{1-\varepsilon}$ then after running the load balancing scheme using a (d, ε, δ)-expander, the maximum number of items in any bucket is bounded by $\frac{kn}{(1-\delta)v} + \log_{(1-\varepsilon)\frac{d}{k}} v$.*

PROOF. Let $B(i)$ denote the number of buckets having *more* than i items, and let μ stand for $\frac{kn}{(1-\delta)v}$. By the pigeonhole principle we have that $B(\mu) < \frac{kn}{\mu} = (1-\delta)v$. We will show that $(1-\varepsilon)\frac{d}{k} \cdot B(\mu+i) \leq B(\mu+i-1)$, for $i \geq 1$. Note that there are at least $B(\mu+i)/k$ left vertices that have placed an item in a bucket of load more than $\mu + i$ (after placement). Denoting the set of such left vertices by S_i, by the expansion property we have

$$|\Gamma(S_i)| \geq \min((1-\varepsilon)d \cdot B(\mu+i)/k,\ (1-\delta)v)\ .$$

Every vertex from S_i has all its neighboring buckets filled with more that $\mu + i - 1$ items, since the vertex was forced to put an item into a bucket of load larger than $\mu + i - 1$. If $|\Gamma(S_i)|$ was not smaller than $(1-\delta)v$ then we would have $B(\mu + i - 1) \geq (1-\delta)v$, which is a contradiction. As a result, $B(\mu+i) < (1-\delta)v \cdot ((1-\varepsilon)\frac{d}{k})^{-i}$, and it follows that $B(\mu + \log_{(1-\varepsilon)\frac{d}{k}} v) = 0$. □

It is not hard to observe that we actually get a bound of $\min(\frac{kn}{q} + \log_{(1-\varepsilon)\frac{d}{k}} q)$, where the minimum is over q ranging from 1 to $(1-\delta)v$. The simpler statement in the lemma is sufficient for the dictionary application, as we use expanders with v not too big.

4. DICTIONARIES ON PARALLEL DISKS

We consider dictionaries over a universe U of size u. It is sufficient to describe structures that support only lookups and insertions into a set whose size is not allowed to go beyond N, where the value of N is specified on initialization of the structure. This is because the dictionary problem is a *decomposable search problem*, so we can apply standard, worst-case efficient global rebuilding techniques (see [12]) to get fully dynamic dictionaries, without an upper bound on the size of the key set, and with support for deletions. The main observations, assuming that we allow the number of disks to increase by a constant factor, are:

- The global rebuilding technique needed keeps two data structures active at any time, which can be queried in parallel.
- We can mark deleted elements without influencing the search time of other elements.
- We can make any constant number of parallel instances of our dictionaries. This allows insertions of a constant number of elements in the same number of parallel I/Os as one insertion, and does not influence lookup time.

The amount of space used and the number of disks increase by a constant factor compared to the basic structure. By the observations above, there is no time overhead. Deletions have the same worst case time bound as insertions.

4.1 Basic dictionary functionality

Without satellite information

Use a striped expander graph G with $v = N/\log N$, and an array of v (more elementary) dictionaries. The array is split across $D = d$ disks according to the stripes of G. The vertices from V represent indexes to the elements of the array. The dictionary implements the load balancing

scheme described above, with $k = 1$. This gives a load of size $\Theta(\log N)$ on each bucket.

If the block size B is $\Omega(\log N)$, the contents of each bucket can be stored in a trivial way in $O(1)$ blocks. Thus, we get a dictionary with constant lookup time. By setting $v = O(N/B)$ sufficiently large we can get a maximum load of less than B, and hence membership queries take 1 I/O. The space usage stays linear.

The constraint $\Omega(\log N)$ is reasonable in many cases. Yet, even without making any constraints on B, we can achieve a constant lookup and insertion time by using an atomic heap [8, 9] in each bucket. This makes the implementation more complicated; also, one-probe lookups are not possible in this case.

With satellite information

If the size of the satellite data is only a constant factor larger than the size of a key, we can increase v by a constant factor to allow that the associated data can be stored together with the keys,and can be retrieved in the same read operation. Larger satellite data can be retrieved in one additional I/O by following a pointer. By changing the parameters of the load balancing scheme to $k = d/2$ and $v = kN/\log N$, it is possible to accommodate lookup of associated information of size $O(BD/\log N)$ in one I/O. Technicalities on one-probe lookups – what exactly to write and how to merge the data – are given in the description of a one-probe static dictionary in Section 4.2

4.2 Almost optimal static dictionary

The static dictionary presented in this section is interesting in its own right: it offers one-probe lookups with good bandwidth utilization, uses linear space when $B = \Omega(\log n)$, and the construction complexity is within a constant factor from the complexity of sorting nd keys, each paired with some associated data. It is not optimal because it uses a bit more space when B is small and the construction procedure takes more time than the time it takes to sort the input, which would be fully optimal. The methods of this dictionary serve as a basis of the dynamic structure of the next section.

From now on, the graph $G = (U, V, E)$ is assumed to be an (N, ε)-expander, unless otherwise stated. We will work only with sets such that $n \leq N$.

Recall that the unique existential quantifier is denoted by $\exists!$ – it can be read as "there exists a unique". Let

$$\Phi_G(S) = \{y \in V \mid (\exists! x \in S)\ (x, y) \in E\}\ .$$

We call the elements of $\Phi_G(S)$ *unique neighbor* nodes.

LEMMA 4. $|\Phi_G(S)| \geq (1 - 2\varepsilon)d|S|$.

PROOF. We define a chain of sets: $T_k = \{x_1, \ldots, x_k\}$, for $1 \leq k \leq n$, as follows:

$$\Gamma(T_{k+1}) \setminus \Gamma(T_k) = \Phi(T_{k+1}) \setminus \Phi(T_k)\ .$$

In the worst case, all the elements of $\Gamma(x_{k+1}) \cap \Gamma(T_k)$ will be in $\Phi(T_k)$. This leads to the inequality

$$|\Phi(T_{k+1})| \geq |\Phi(T_k)| - d + 2(|\Gamma(T_{k+1})| - |\Gamma(T_k)|)\ .$$

By induction, for all $k \leq n$ it holds that

$$2|\Gamma(T_k)| - |\Phi(T_k)| \leq k \cdot d\ .$$

Therefore $|\Phi(S)| \geq 2(1 - \varepsilon)nd - nd$. □

LEMMA 5. *Let $S' = \{x \in S \mid |\Gamma(x) \cap \Phi(S)| \geq (1 - \lambda)d\}$, for a given $\lambda > 0$. Then $|S'| \geq (1 - \frac{2\varepsilon}{\lambda})n$.*

PROOF. W.l.o.g. suppose $S' = \{x_{k+1}, x_{k+2}, \ldots, x_n\}$. Let k^* be the largest integer that satisfies $k^*(1-\lambda)d+(n-k^*)d \geq |\Phi(S)|$. It is easy to see that $k^* \geq k$. Using Lemma 4 gives $k^* \leq \frac{2\varepsilon}{\lambda}n$. □

For the following static and dynamic dictionary results, the stated numbers of used disks represent the minimum requirement for functioning of the dictionaries. The record of one key, together with some auxiliary data, is supposed to be distributed across $\frac{2}{3}d$ disks. For 1 I/O search to be possible, every distributed part of the record must fit in one block of memory. If the size of the satellite data is too large, more disks are needed to transfer the data in one probe. The degree of the graph does not change in that case, and the number of disks should be a multiple of d. Recall that we assume availability of a suitable expander graph construction such that $d = O(\log u)$.

THEOREM 6. *Let σ denote the size in bits of satellite data of one element, and let d be the degree of the given (n, ε)-expander graph. In the parallel disk model there is a static dictionary storing a set of n keys with satellite data, such that lookups take one parallel I/O and the structure can be constructed deterministically in time proportional to the the time it takes to sort nd records with $\Theta(n\sigma)$ bits of satellite information in total. The exact usage of resources depends on the block size relative to the size of a key:*

a) If $O(\log n)$ keys can fit in one memory block, then the structure uses $2d$ disks and a space of $O(n(\log u + \sigma))$ bits;

b) If the block size is smaller, then d disks are used and the space consumption is $O(n \log u \log n + n\sigma)$ bits.

The space usage in case (a) is optimal, up to a constant factor, when u is at least polynomially larger than n. When the universe is tiny, a specialized method is better to use, for example simple direct addressing. The space usage in case (b) is optimal when $\sigma > \log u \log n$.

PROOF. Fix a value of ε that will always satisfy $1/d < \varepsilon < 1/6$; for concreteness we set $\varepsilon = 1/12$ (this imposes the restriction $d > 12$). The data structure makes use of a striped (n, ε)-expander graph of left degree d and with $v = O(nd)$. The main data is stored in an array A of v fields, where the size of a field depends on the case – (a) or (b). We will first explain the structure for case (b) in detail, then we will describe modifications made to optimize the space usage in case (a), and finally give the algorithm for construction of the structures.

Structure in case (b). Every field of A has size $\lg n + \frac{3\sigma}{2d}$ bits (possibly rounded up to the nearest power of two). Given any $x \in U$ the set $\Gamma(x)$ is viewed as the set of indexes to the fields of A that *may* contain data about x. We will later describe how to accomplish that, for every $x \in S$, a fraction $2/3$ of the fields referenced by $\Gamma(x)$ store parts of data about the associated record for x. When a lookup is performed for a key x, all the fields of A pointed to by $\Gamma(x)$ are read into the internal memory in one parallel I/O. However, not all of them store data belonging to x. Deciding the correct

ones could be done by storing a copy of the key in each of the fields of A that were assigned to it. Yet, a more space efficient solution is to use identifiers of $\lg n$ bits, unique for each element of S. Upon retrieval of the blocks from disks, it is checked whether there exists an identifier that appears in more than half of the fields. If not, then clearly $x \notin S$, otherwise the fields containing the majority identifier are merged to form a record of associated data. Note that no two keys from U can have more than εd common neighbors in V. Therefore, we know that the collected data belongs to x – there is no need for an additional comparison, or similar.

Structure in case (a). When the block size is reasonably large, we can avoid storing $\lg n$ bits wide identifiers within fields of A. We use two sub-dictionaries in parallel – one for pure membership queries and another for retrieval of satellite data. Half of the $2d$ available disks is devoted to each dictionary. Querying membership in S is done using the dictionary given in Section 4.1. Every stored key is accompanied by a small integer of $\lg d$ bits, which we call its *head pointer*. By the assumption of the theorem for this case, $O(\log n)$ such key-pointer pairs can fit in one block, thereby enabling one probe queries according to Section 4.1.

The retrieval structure is similar to the dictionary for the case (b). The array A now has fields of size $\frac{3\sigma}{2d} + 4$ bits. Instead of "big" identifiers we choose to store succinct pointer data in every field; the fraction of an array field dedicated to pointer data will vary among fields. For every x we may introduce an ordering of the neighbor set $\Gamma(x)$ according to the stripes of V. That order implies an order of the fields of A assigned to a particular element. Each assigned field stores a *relative* pointer to the next field in the list – if the jth neighbor follows the ith neighbor in the list of assigned nodes then the value $j - i$ is stored within $A(\Gamma(x, i))$, where $\Gamma(x, i)$ denotes the ith neighbor of x. The differences are stored in unary format, and a 0-bit separates this pointer data from the record data . The tail field just starts with a 0-bit. The entire space occupied by the pointer data is less than $2d$ bits per element of S; all the remaining space within fields is used on storing record data. Upon parallel retrieval of blocks from both dictionaries, we first check whether x is present in S. If it is, we use the head pointer to reconstruct the list and merge the satellite data.

Construction in $O(n)$ I/Os. Assigning $\lfloor \frac{2}{3}d \rfloor$ neighbors to each key from S is done using the properties of unique neighbor nodes. By setting $\lambda = 1/3$, according to Lemma 5 and the choice of ε, at least half of the elements of S have at least $\frac{2}{3}d$ neighbors unique to them. For each $x \in S'$ (where S' is defined in Lemma 5) any $\frac{2}{3}d$ unique neighbors are chosen to store its satellite data; the fields of unused nodes from $\Gamma(S') \cap \Phi(S)$ are labeled with an empty-field marker. The entire process of determining $\Phi(S)$, S', and filling the fields can be done in less than $c \cdot n$ parallel I/Os, for a constant c. The procedure is recursively applied to the set $S \setminus S'$, independently of the assignments done at the first level, because there is no intersection between the assigned neighbor set for S' and $\Gamma(S \setminus S')$. The whole assignment procedure takes less than $c(n + n/2 + n/4 + \ldots) = O(n)$ I/Os.

Improving the construction. We keep the concept of making assignments using unique neighbor nodes, but change the procedure that realizes it. We assume that the input has a form of an array of records split across the disks, but with individual records undivided (this should be a standard representation). We will describe a procedure that is first applied to the input array, and then recursively applied to the array obtained by removing the elements of S' from the input array. Unlike the first version of the algorithm, the job is not completely finished for the elements of S' at the end of the procedure. The ouput of the procedure is an array of pairs of type (i, α_i), where α_i is data that is to be written to $A(i)$. This array will contain information only about the fields assigned to the elements of S'. Each time the procedure finishes execution, the output is appended to a global array, call it B.

The procedure starts by making an array of all pairs of type (x, y), $x \in S$, $y \in \Gamma(x)$. This array is sorted according to the second component of pairs, and then traversed to remove all sequences of more than one element that have equal values of the second components. This effectively leaves us a list of unique neighbor nodes, each paired with the (only) neighbor from the left side. By sorting this list of pairs according to the fist components, we get the elements of $\Phi(S) \cap \Gamma(x)$ grouped together, for each $x \in S$, and we are able to remove data about members of S that do not have enough unique neighbors. We now have a list of the elements of S', with associated lists of unique neighbors.

We sort the input array of records according to the dictionary keys. The resulting array is traversed simultaneously with the array of elements of S' (recall that this array is also sorted), allowing us to produce an array of pairs of type (i, α_i). The description of the main procedure is now finished.

When the contents of the array B is final (at the end of the recursion), it is sorted according to the first components of elements; this is the most expensive operation in the construction algorithm. Filling the array A is at this point a straightforward task. □

4.3 Full bandwidth with $1 + \epsilon$ average I/O

The above static dictionary is not hard to dynamize in a way that gives fast average-case lookups and updates. We concentrate on the case (a) from Theorem 6. A slightly weaker result is possible in the more general case as well. In this section, the reader should be careful to distinguish between symbols ϵ and ε; ϵ is a parameter of operation performance, while ε is a parameter for expander graphs and its value will depend on the value of ϵ.

THEOREM 7. *Let ϵ be an arbitrary positive value, and choose d, the degree of expander graphs, to be (a feasible value) larger than $6(1 + 1/\epsilon)$. Under the conditions of Theorem 6.a, there is a deterministic dynamic dictionary that provides the following performance: an unsuccessful search takes one parallel I/O, returning the associated data when a search is successful takes $1 + \epsilon$ I/Os averaged over all the elements of S, updates run in $2 + \epsilon$ I/Os on average.*

PROOF. As mentioned at the beginning of this section it is enough to describe a structure that supports only lookups and insertions into a set whose size is not allowed to go beyond N, where the value of N is specified on initialization of the structure. As in the static case, we use two sub-dictionaries. Since the first dictionary is already dynamic,

modifications are made only to the dictionary that retrieves associated data. We choose ε so that $\frac{6}{d} < 6\varepsilon < 1/(1+\frac{1}{\epsilon})$; the restriction on d was imposed to make this possible. Instead of just one array, now there are $l = \log N/\log\frac{1}{6\varepsilon}$ arrays of decreasing sizes: $A_1, A_2, \ldots, A_l$. The size of the array A_i is $(6\varepsilon)^{i-1}v$. Each array uses a different expander graph for field indexing – all expander graphs have the same left set U, the same degree d, but the size of each expander's right side equals the size of its corresponding array.

The insertion procedure works in a first-fit manner: for a given $x \in U$ find the first array in the sequence $(A_1, A_2, \ldots, A_l)$ in which there are $\frac{2}{3}d$ fields unique to x (at that moment). Using Lemma 5, it is not hard to check that the procedure is correct, i.e. a suitable place is always found. To briefly argue this observe that for the resulting set of an insertion sequence, denote it by S, A_1 will hold the data for a superset of S' (where S' is defined as in Lemma 5 and with respect to the first expander graph), and so on. In the worst case an insertion takes l reads and one write. However, any sequence of n insertions, $n \leq N$, requires n parallel writes and less than

$$n + (6\varepsilon)n + (6\varepsilon)^2 n + \ldots + (6\varepsilon)^l n$$

parallel read operations. The choice of ε implies the average of less than $1 + \epsilon$ reads. □

5. EXPLICIT CONSTRUCTIONS

In the previous sections we have assumed "free" access to an explicit optimal expander. An expander construction (and any expander graph constructed by it) is considered explicit if one can evaluate $\Gamma(x)$, for a left vertex x, in time polylog(u), given the size of U and V. It is not known how to obtain an optimal explicit expander, when $u = \omega(N)$.

In the context of the external memory model it makes sense to allow an expander construction to make use of a small amount of internal memory, the primary issue being avoiding access to external memory in order to evaluate the neighbors of a left vertex. In this section we therefore consider what we will call semi-explicit expander constructions, which use $o(N)$ words of internal memory and are allowed a pre-processing step, but which still allows the neighbors of a left vertex to be evaluated in time polylog(u) with no access to external memory.

We first discuss previous related results and then show how to achieve a semi-explicit expander construction requiring, for any constant $\beta > 0$, $O(N^\beta)$ words of internal memory, in the case where $u = \text{poly}(N)$.

5.1 Previous Results

For the case of balanced expanders with arbitrarily small ε, the best known explicit expander is due to Capalbo et al. [6] who achieve an expander with constant degree when the graph is almost balanced, i.e., $\frac{u}{v} = O(1)$. In the case of arbitrarily unbalanced expanders the best known result is due to Ta-Shma:

THEOREM 8 ([17]). *For any N, u, and ε, where $\varepsilon < 1$ and $N \leq u$, there exists an explicit construction of an (N, ε)-expander graph defined by $F : [u] \times [d] \to [Nd]$, of degree $d = 2^{O((\log\log u)^2 \log\log N)}$*

Expanders requiring some pre-processed storage space were previously considered in [16]. However the representation of the resulting expanders require space $\Omega(N)$, and are therefore not suitable for expander based dictionaries where the expander representation is stored in internal memory.

5.2 Construction

The following result from [6, Theorem 7.1] plays an important role in our construction:

THEOREM 9 ([6]). *For every u and $\varepsilon > 0$, there exists an (N, ε)-expander defined by $F : U \times [d] \to V$, where $v = O(Nd)$, $d = \text{poly}\left(\frac{1}{\varepsilon}\log\frac{u}{v} + 1\right)$, and $N = \Theta\left(\frac{v\cdot\varepsilon}{d}\right)$.*

F can be computed in time poly$(\log u, \log\frac{1}{\varepsilon})$ given two appropriate expanders using $s = \text{poly}(\frac{u}{v}, \frac{1}{\varepsilon})$ bits, which can be found probabilistically in time poly(s) or deterministically in time $2^{poly(s)}$.

If we restrict our attention to the case where $N = u^\alpha$ where $0 < \alpha < 1$ the space usage of poly$(\frac{u}{v}, \frac{1}{\varepsilon})$ becomes poly$(\frac{u}{u^\alpha d}, \frac{1}{\varepsilon})$ = poly$(\frac{u^{1-\alpha}}{d}, \frac{1}{\varepsilon})$ which cannot be guaranteed to be $o(N)$. However for a certain range of α it is indeed possible to obtain a space usage of $o(N)$ as we will see below.

COROLLARY 1. *For any constant $0 < \beta < 1$ there exists a semi-explicit $(\frac{\varepsilon}{d}u^{1-\beta/c}, \varepsilon)$-expander $F : U \times [d] \to [u^{1-\beta/c}]$ constructed using $O(u^\beta/\varepsilon^c)$ words of space, where $d = \text{poly}(\frac{\log u}{\varepsilon})$ and c is some fixed constant.*

PROOF. The construction in Theorem 9 requires $O((\frac{u}{v\varepsilon})^c)$ space, for some constant c. We wish to use $O(u^\beta/\varepsilon^c)$ space. We set u/v as small as possible while maintaining this space usage that is $u/v = u^{\beta/c}$. This yields $v = u^{1-\beta/c}$. Observing that the remaining required properties are given by Theorem 9 ends the proof. □

We can use Corollary 1 in conjunction with a composition that allows us to use two slightly unbalanced expanders to produce a more unbalanced expander. We will refer to this composition method, previously used on condensers by Ta-Shma et al. in [18], as the *telescope product*.

LEMMA 10. *Let c_1, c_2 be constants where $c_1 \geq c_2$ and let $F_1 : U_1 \times [d_1] \to V_1$ be a $(\frac{c_1\cdot v_1}{d_1}, \varepsilon_1)$-expander and let $F_2 : V_1 \times [d_2] \to V_2$ be a $(\frac{c_2\cdot v_2}{d_2}, \varepsilon_2)$-expander. Then for $x_1 \in U_1$, $e_1 \in [d_1]$ and $e_2 \in [d_2]$, the function*

$$F_2(F_1(x_1, e_1), e_2) : U_1 \times ([d_1] \times [d_2]) \to V_2$$

with appropriate re-mapping of possible multi-edges is a $(\frac{c_2\cdot v_2}{d_1\cdot d_2}, 1 - (1-\varepsilon_1)(1-\varepsilon_2))$-expander.

PROOF. Consider any set $S \subseteq U_1$ of size $s \leq \frac{c_2\cdot v_2}{d_1\cdot d_2}$. This set is small enough to be fully expanded by F_1, since $\frac{c_2\cdot v_2}{d_1\cdot d_2} \leq \frac{c_1\cdot v_1}{d_1}$ by the assumption that $c_2 \leq c_1$. The expansion of S hence yields a set of neighbors $V' \subseteq V_1$ of size

$$v' \geq (1-\varepsilon_1)\cdot\frac{c_2\cdot v_2}{d_2} \ .$$

This set is small enough to be fully expanded by F_2, yielding a set of neighbors $V'' \subseteq V_2$ of size

$$v'' \geq (1-\varepsilon_2)(1-\varepsilon_1)\cdot v_2\cdot c_2 \ .$$

The result of this composition may be a multi-graph. In order to rectify this, we define the following neighbor function: When evaluating a neighbor of $x \in U_1$ we evaluate all neighbors of x in v_2 and re-map all but one edge in each multi-edge in an appropriate and fixed manner. This cannot decrease the expansion factor. □

The need to evaluate all neighbors does not affect the time complexity of the dictionaries in this paper, since we always evaluate all neighbors. When only evaluating a single neighbor this increases the time complexity by a factor $d_1 d_2$.

LEMMA 11. *For any $\beta' < c$, any $\varepsilon' > 0$, and any $j \geq 0$ there exist an $(\varepsilon' u_j/d_j, 1-(1-\varepsilon')^{j+1})$-expander $F^{(j)} : [u] \times [d_j] \to [u_j]$ using $O(j \cdot u^{\beta'}/\varepsilon'^c)$ words of internal memory and requiring time poly(d_j) to evaluate neighbors, of degree $d_j = (poly(\log(u)/\varepsilon'))^{j+1}$, and right part of size $u_j = u^{(1-\beta'/c)^j}$.*

PROOF. Consider the following family of expanders obtained by Corollary 1:

$$F_{i,\varepsilon'} : [u_i] \times [\text{poly}(\log(u)/\varepsilon')] \to [u_{i+1}]$$

Such that each $F_{i,\varepsilon'}$ is an $(\frac{\varepsilon' u_{i+1}}{\text{poly}(\log(u)/\varepsilon')}, \varepsilon')$-expander, where $u_i = u^{(1-\beta'/c)^i}$. Now consider the telescope product applied recursively on this family:

$$F^{(j)}(x, e_1 \circ e_2 \circ ...e_j) = \begin{cases} F_{j,\varepsilon'}(F^{(j-1)}(x, e_1 \circ e_2 \circ ...e_{j-1}), e_j) & \text{if } j > 0 \\ F_{0,\varepsilon'}(x, e_1) & \text{otherwise} \end{cases}$$

By induction using Lemma 10 we see that $F^{(j)}$ satisfies the lemma. The complexity of evaluating neighbors is just the time to evaluate the neighbors of neighbors in each of the expanders in the family, and is easily seen to be bounded by poly(d_j). □

THEOREM 12. *For any constant $0 < \beta < 1$ and $u = poly(N)$ there exists a semi-explicit (N, ε)-expander defined by $F : U \times [d] \to V$, with $d = polylog(u)$ and $v = O(Nd)$ requiring $O(N^\beta)$ words of pre-processed internal memory.*

PROOF. (For full details we refer to [4].) Let $u = O(N^{1/\alpha})$. We note that for sufficiently high i, $F^{(i)}$ from Lemma 11 reaches a point where $v \leq Nd_i$. Let $p = \frac{1}{1-\beta/c}$, then the required number of iterations k is bounded by

$$\log_p(\log u/\log N) + O(1) = \log_p(a) + O(1) = O(1) \ .$$

To obtain a total error of ε we set $\varepsilon' = 1 - \sqrt[k]{1-\varepsilon}$ such that $(1-\varepsilon) = (1-\varepsilon')^k$. Finally, by setting $\beta' = \alpha \cdot \beta$ we ensure a space usage of $O(k \cdot u^{\alpha\beta}/\varepsilon'^c) = O(N^\beta/\varepsilon'^c)$. □

We have now attained a new construction of an unbalanced expander, when u is polynomial in N. Our construction requires a degree of $(\log N)^{O(1)}$, compared to the previous best result of $(\log N)^{O((\log\log N)^2)}$, by using some internal memory. Like all mentioned explicit expander constructions, our construction does not yield a striped expander. If we implement the described dictionaries in the parallel disk head model, we do not need the striped property.

To get an algorithm for the parallel disk model we may stripe an expander $F : U \times [d] \to V$ in a trivial manner by making a copy V_i of the right side V of the expander for each disk i. In order to find the neighbor of $x \in U$, we calculate $F(x, i)$ and return the corresponding vertex in V_i. This incurs a factor d increase in the size of the right part of the expander, and hence a factor d larger external memory space usage.

6. OPEN PROBLEMS

It is plausible that full bandwidth can be achieved with lookup in 1 I/O, while still supporting efficient updates. One idea that we have considered is to apply the load balancing scheme with $k = \Omega(d)$, recursively, for some constant number of levels before before relying on a brute-force approach. However, this makes the time for updates non-constant. It would be interesting if this construction could be improved.

Obviously, improved expander constructions would be highly interesting in the context of the algorithms presented in this paper. It seems possible that practical and truly simple constructions could exist, e.g., a subset of d functions from some efficient family of hash functions.

7. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[3] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: the heavily loaded case. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 745–754. ACM Press, 2000.

[4] M. Berger, E. R. Hansen, and P. Tiedemann. Expander based dictionary data structures. Master's thesis, IT-University of Copenhagen, 2005. Available at: http://www.itu.dk/people/esben/publications/thesis.pdf.

[5] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 449–458. ACM Press, 2000.

[6] M. R. Capalbo, O. Reingold, S. P. Vadhan, and A. Wigderson. Randomness conductors and constant-degree lossless expanders. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*, pages 659–668, 2002.

[7] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.

[8] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.

[9] T. Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[10] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

[11] A. Östlin and R. Pagh. One-probe search. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*, volume 2380 of *Lecture Notes in Computer Science*, pages 439–450. Springer, 2002.

[12] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inform. Process. Lett.*, 12(4):168–173, 1981.

[13] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[14] J. P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC '90)*, pages 224–234. ACM Press, 1990.

[15] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995.

[16] A. Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. Comput.*, 33(3):505–543, 2004.

[17] A. Ta-Shma. Storing information with extractors. *Inf. Process. Lett.*, 83(5):267–274, 2002.

[18] A. Ta-Shma, C. Umans, and D. Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 143–152. ACM Press, 2001.

[19] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, Aug./Sept. 1994.