Exponential Structures for Efficient Cache-Oblivious Algorithms*

Michael A. Bender[1], Richard Cole[2], and Rajeev Raman[3]

[1] Computer Science Department, SUNY Stony Brook, Stony Brook, NY 11794, USA. bender@cs.sunysb.edu.

[2] Computer Science Department, Courant Institute, New York University, New York, NY 10012, USA. cole@cs.nyu.edu.

[3] Department of Mathematics and Computer Science, University of Leicester, Leicester LE1 7RH, UK. R.Raman@mcs.le.ac.uk

Abstract. We present *cache-oblivious* data structures based upon *exponential structures*. These data structures perform well on a hierarchical memory but do not depend on any parameters of the hierarchy, including the block sizes and number of blocks at each level. The problems we consider are searching, partial persistence and planar point location. On a hierarchical memory where data is transferred in blocks of size B, some of the results we achieve are:

– We give a linear-space data structure for dynamic searching that supports searches and updates in optimal $O(\log_B N)$ *worst-case* I/Os, eliminating amortization from the result of Bender, Demaine, and Farach-Colton (FOCS '00). We also consider finger searches and updates and batched searches.

– We support partially-persistent operations on an ordered set, namely, we allow searches in any previous version of the set and updates to the latest version of the set (an update creates a new version of the set). All operations take an optimal $O(\log_B(m+N))$ amortized I/Os, where N is the size of the version being searched/updated, and m is the number of versions.

– We solve the planar point location problem in linear space, taking optimal $O(\log_B N)$ I/Os for point location queries, where N is the number of line segments specifying the partition of the plane. The pre-processing requires $O((N/B)\log_{M/B} N)$ I/Os, where M is the size of the 'inner' memory.

1 Introduction

A modern computer has a hierarchical memory consisting of a sequence of levels – machine registers, several levels of on-chip cache, main memory, and disk – where the farther the level is from the CPU, the larger the capacity and the longer the access time. On present-day machines, the cache is 2 orders of magnitude faster than main memory, and 6 - 7 orders of magnitude faster than disk. In order to amortize the cost of a memory access, data is transferred between levels in *blocks* of contiguous locations; we refer to each such transfer as an I/O. Due to the cost of an I/O, it is important to minimise I/Os.

* This work was supported in part by HRL Laboratories, NSF Grant EIA-0112849 and Sandia National Laboratories (Bender); in part by NSF grants CCR-9800085 and CCR-0105678 (Cole); and in part by EPSRC grant GR L/92150 and UISTRF project 2001.04/IT (Raman).

Unfortunately, a multilevel memory hierarchy is complex, and it can be unwieldy or impossible to write programs tuned to the parameters of each level of the memory hierarchy. Also, this parameterization leads to inflexible algorithms that are only tuned for one memory platform. The traditional alternative is to assume a two-level memory rather than a multilevel memory. The best-known two-level memory model was defined by Aggarwal and Vitter [1] (see [24] for others). In this model the memory hierarchy is composed of an arbitrarily large disk divided into blocks, and an internal memory. There are three parameters: the problem size N, the block size B, and the main memory size M. An I/O transfers one block between main memory and disk. The goal is to minimize the number of I/Os, expressed as a function of N, M, and B. In recent years many algorithms have been designed for the Aggarwal-Vitter and related models, many of which are reviewed in some recent surveys [24,5].

One limitation of the Aggarwal-Vitter model is that it only applies to two-level memory hierarchies. In contrast, it is becoming increasingly important to achieve data locality on *all* levels of the hierarchy. For example, Ailamaki et al. [2] report that in many standard database applications, the bottleneck is main-memory computation, caused by poor cache performance. Even in main memory, simultaneous locality at cache and disk block levels improves performance by reducing address translation misses [19,20]. Furthermore, the memory hierarchy is rapidly growing steeper because in recent years, processing speeds have been increasing at a faster rate than memory speeds (60% per year versus 10% per year). This trend is entrenched and appears unlikely to change in the next few years if not longer.

Frigo et al. [16,18] proposed the elegant *cache-oblivious model*, which allows one to reason about a two-level memory hierarchy, but prove results about an unknown multilevel memory hierarchy. Cache-oblivious algorithms assume a standard (flat) memory as in the RAM model, and are unaware of the structure of the memory hierarchy, including the capacities, block sizes, or access times of the levels. I/Os occur transparently to the algorithm, and I/O performance is analyzed in the Aggarwal-Vitter model [1] with arbitrary block and main memory sizes. The model assumes an *ideal cache*, where I/Os are performed by an omnisicent off-line algorithm (Frigo et al. note that any reasonable block-replacement strategy approximates the omniscient strategy to within a constant factor). The key observation is that if the algorithms perform few I/Os in this scenario, then they perform well for all block and main memory sizes on all levels of the memory hierarchy simultaneously, for any memory hierarchy. By way of contrast, henceforth we refer to the Aggarwal-Vitter model as the *cache-aware* model.

Frigo et al. [16,18] gave optimal algorithms for sorting, FFT, and matrix transpose. Optimal cache-oblivious algorithms have also been found for LU decomposition [11, 23]. These algorithms make the (generally reasonable) *tall cache* assumption that $M = \Omega(B^2)$. Prokop [18] observed how to layout a complete binary tree in linear space so as to optimize root to leaf traversals in the I/O cost model. We call his layout a *height partitioning* layout, and review it later. Bender, Demaine and Farach-Colton [8] considered the dynamic predecessor searching problem, and obtained a linear space data structure with amortized $O(\log_B N)$ update cost and worst case $O(\log_B N)$ search cost. They also consider the problem of traversing the leaves of the tree. Their result, like most of our results, does not use the tall cache assumption.

Our approach. In this paper we present a new general approach for solving irregular and dynamic problems cache-obliviously: *the exponential structure,* which is based on the exponential tree previously developed and used by Andersson and Thorup [3,22,4] for designing fast data structures in the transdichotomous model [15]. An exponential tree is a tree of $O(\log \log N)$ levels where the degrees of nodes descending from the root level decrease doubly exponentially, e.g. as in the series $N^{1/2}, N^{1/4}, N^{1/8}, \cdots, 2$.

We show that exponential trees, and more generally exponential structures, are similarly powerful in a hierarchical memory. We use exponential trees to devise alternative solutions to the cache-oblivious B-tree, and we provide the first worst-case solution to this problem. Then we generalize these techniques to devise new cache-oblivious data structures, such as persistent search trees (which are not trees), search trees that permit finger searches, search trees for batched searches, and a planar point location structure. By using exponential structures to solve a variety of problems, we introduce a powerful tool for designing algorithms that are optimized for hierarchical memories.

Our results are as follows:

Dynamic predecessor searching: The problem here is to store a set of N totally ordered items, while supporting predecessor queries, insertions and deletions. We give a series of data structures, culminating in one that takes $O(N)$ space and supports all operations in $O(\log_B N)$ worst-case I/Os, which is optimal. This eliminates amortization from previous cache-oblivious solutions to this problem [8,12,10].

The main difficulty faced in supporting dynamic search structures cache-obliviously is the need to maintain data locality (at all levels of granularity) in the face of updates. Arbitrary updates appear to require data sparsity. However, sparsity reduces locality. To balance these constraints, Bender et al. [8] developed strongly weight-balanced trees and the *packed-memory* structure. Two other recent solutions with the same amortized complexity also use the packed memory structure or similar techniques [12,10]. In essence, the packed-memory problem is to maintain an ordered list of length n in an array of size $O(n)$, preserving the order, under insertions and deletions. Although simple amortized algorithms to the packed-memory problem in the RAM model yield good cache-oblivious algorithms, it is far from clear if the same holds for Willard's complex worst-case RAM solution [25].

As we will see, exponential structures are particularly effective at addressing the tension between locality and sparsity. Our solution uses a new worst-case approach to dynamizing exponential trees. In contrast to an earlier worst-case dynamization by Andersson and Thorup [4] in the context of dynamic searching in the RAM model, the main difficulty we face is to balance data sparsity and locality for dynamically changing "fat" nodes in the exponential tree. As a result, our solutions, unlike Andersson and Thorup's, will use superlinear space at the deeper levels of the search structure. One of our dynamizations seems to be simpler than that of Andersson and Thorup, and we believe would somewhat simplify their constructions.

Finger search: This is the variant of dynamic predecessor searching in which successive accesses are nearby. More precisely, suppose two successive accesses to an ordered set are separated by d elements. We give a linear size data structure such that the second access performs $O(\log^* d + \log_B d)$ amortized and worst-case I/Os respectively, depending on whether it is an update or a search.

By way of comparison, in the standard RAM model there are solutions using $O(\log d)$ operations [13,17], and this extends in a straightforward way to the cache-aware model with bounds of $O(\log_B d)$. The cache-oblivious data structures of [8,10] do not support finger searches, and that of [12] has a higher update time.

Partial persistence: Here one seeks to support queries in the past on a changing collection of totally ordered items. Specifically, suppose we start with an ordered set of N items at time t_0, and then perform a series of m updates at times $t_1 < t_2 < \cdots < t_m$ ($t_0 < t_1$, of course). At any time t'_j, $t_j < t'_j < t_{j+1}$, one can perform a query of the form: "what was the largest item $x < y$ at time t_i?", where $i \leq j$. We support updates and queries in $O(\log_B(N + m))$ I/Os and use $O(N + m)$ space. Note that the structure here is a dag. A key issue is to give an appropriate definition of the weight of a fat node.

By way of comparison we note that in the standard RAM model there are solutions using $O(\log(N + m))$ operations [21], and our I/O bound matches the cache-aware result of [14].

Planar point location: The input comprises N arbitrarily-oriented line segments in the plane which are non-intersecting, except possibly at the endpoints. The task is to preprocess the line segments so as to rapidly answer queries of the form: which line segment is directly above query point p? (The traditional planar point location problem boils down to solving this problem.)

This can be solved using a structure for partial persistence that supports predecessor queries on items drawn from a partially ordered universe, but with the requirement that the items present at any given time t_i be totally ordered; however, there is no requirement that items have a relative ordering outside their time span of existence. Sarnak and Tarjan [21] gave a linear space solution for this problem supporting queries in $O(\log N)$ time. As our partial persistence data structure does not support partial orders, we are obliged to take a different approach. We build a search dag similar to that used for our partial persistence data structure, but build it offline; it has linear size, is built using $O((N/B) \log_{M/B} N)$ I/Os, and supports queries using $O(\log_B N)$ I/Os. This is a worst case result, and depends on the tall cache assumption. This data structure also supports batched queries efficiently.

By way of comparison, several linear-space data structures are known in the RAM model that support queries in $O(\log N)$ time and can be constructed in $O(N \log N)$ time. In the two-level model, Goodrich et al. [14] give a linear space data structure that supports queries in $O(\log_B N)$ I/Os but whose construction takes $O(N \log_B N)$ I/Os. Arge et al. [6] consider a batched version of this problem, but their data structure requires superlinear space.

Batched search: We give a linear-sized structure for performing simultaneous predecessor queries on r query items among N base items. This is challenging when there is only a partial order on the r items (so that they cannot necessarily be sorted); however, the N base items are totally ordered and each query item is fully ordered with respect to the base items. (An example scenario has as base items non-intersecting lines crossing a vertical strip in the plane, and as query items points in the vertical strip.) The I/O cost for the search is $O(r \log_B(N/r) + (r/B) \cdot \log_{M/B} r)$. This result uses the tall cache assumption (and hence $\log_{M/B} r = O(\log_M r)$). We give an off-line algorithm for constructing the search structure; it uses $O((N/B) \log_{M/B} N)$ I/Os. We believe we can also maintain the structure dynamically with the same I/O cost as for searches, but have yet to check all the details.

In the rest of this abstract we outline our solutions to some of the above problems. Proofs of the results on batched searching and finger searching, persistent search, as well as details of the strongest version of the planar point location result, are omitted from this abstract and may be found in [9].

2 Dynamic Searching

We do not discuss deletions, which can essentially be handled by standard lazy approaches. Our solution uses a known optimal layout [18] for complete binary search trees as a building block. This layout places an N node tree in a size N array as follows. The tree is partitioned into height $\frac{1}{2}\log N$ subtrees, each of which is laid out recursively in an array segment of length $\sqrt{N}$. We call this the *height-partitioning* layout. It is straightforward to extend this construction to trees in which all the leaves are at the same level, where this is the first level in which this number of leaves would fit. The search time is unchanged and the space is still linear.

Lemma 1. *(Prokop). A search on a tree stored in a height-partitioning layout uses $O(\log_B N)$ I/Os.*

Our dynamic solution uses a similar structure, but in order to handle updates efficiently we need something more flexible. In our search trees, as with B-trees, all items are at the leaves and only keys are stored at internal nodes. As suggested in the introduction, internal nodes may have many children; to emphasize their large size we will call them *fat nodes* henceforth. Likewise, we will refer to a *layer* of fat nodes rather than a level.

We define the *volume* of a fat node in an exponential tree to be the number of items stored in its descendant leaves. We extend the definition to subtrees: the volume of a subtree is simply the volume of its root. Let T be a layer i fat node. An update that changes T's volume will be called an update in T. Note that an update will be in one fat node in each layer.

2.1 The Amortized $O(\log_B N + \log\log N)$ Solution

We parameterize the fat nodes by layer, the leaves forming layer 0. A layer i fat node, $i \geq 1$, will have a volume in the range $[2^{2^i} - 2^{2^{i-1}}, 2 \cdot 2^{2^i})$, except for the topmost fat node, where the range is given by $[2 \cdot 2^{2^{k-1}}, 2 \cdot 2^{2^k})$. The reason for the term "$-2^{2^{i-1}}$" will become clear later. Each layer 0 fat node contains a single item. Loosely speaking, the volumes of the fat nodes square at each successive layer.

Each fat node is implemented as a complete binary tree with all its leaves on the same level; appropriate keys are stored at the leaves and internal vertices. This binary tree is then stored in a height partitioning layout. Each leaf of this binary tree, except perhaps the rightmost one, will have two pointers, one to each of its two children fat nodes. We note that a layer i fat node, $i \geq 1$, has $O(2^{2^{i-1}})$ leaves and internal vertices.

Lemma 2. *A search takes $O(\log_B N + \log\log N)$ I/Os.*

Proof. Going from one layer to another takes one I/O, and there are $\log\log N$ such transfers in all. The searches within layer i consume $O(\log_B 2^{2^{i-1}})$ I/Os. Summed over all the layers, this comes to $O(\log_B N + \log\log N)$. □

Next we explain when a fat node needs updating and how it is done. If a layer i fat node T acquires volume $2 \cdot 2^{2^i}$ it is split as evenly as possible into two subtrees T_1 and T_2, each of size roughly 2^{2^i}. However, we wish to perform this split without changing T's children fat nodes. As the children have volumes up to $2 \cdot 2^{2^{i-1}}$, we can only achieve a split with the subtrees T_1 and T_2 having sizes in the range $[2^{2^i} - 2^{2^{i-1}}, 2^{2^i} + 2^{2^{i-1}}]$. Such a split can readily be implemented in $O(|T|) = O(2^{2^{i-1}})$ time and I/Os.

When T splits this adds one to the number of its parent's children. This is accommodated by completely rebuilding the parent, which can be done in time and I/Os linear in the size of the parent, i.e. $O(2^{2^i})$ time and I/Os. We call this operation *refreshing* the parent. A standard argument shows that the amortized times and I/Os spent by an insertion on splitting and refreshing the layer i fat node T it traverses are $O(1/2^{2^{i-1}})$ and $O(1)$ respectively. Thus, we obtain:

Theorem 1. *There is a cache oblivious linear size data structure for an ordered set of N items supporting searches in worst case $O(\log_B N + \log\log N)$ I/Os and updates in amortized $O(\log_B N + \log\log N)$ I/Os.*

The simplicity and consequent practicality of the above solution has been confirmed in preliminary experimental work [20].

2.2 The Amortized $O(\log_B N)$ I/O Solution

The main change here is to keep a level i layer fat node and all its descendant layer fat nodes in a contiguous portion of the array. This will increase the splitting time and as we will see increases the space needed.

Now when splitting layer i fat node T into subtrees T_1 and T_2, besides creating T_1 and T_2 we need to copy all of T's descendant fat nodes into either the portion of the array being used for T_1 and its descendants or that being used for T_2 and its descendants. This will take time proportional to the size of T and its descendants. Thus the cost of splitting T becomes $O(2^{2^i})$. However, amortized time and I/Os spent by an insertion on splitting the layer i fat node T it traverses is still $O(1)$. Let us turn to the space needed.

Lemma 3. *Layer i fat node T has at most $2 \cdot 2^{2^i}/(2^{2^{i-1}} - 2^{2^{i-2}})$ subtrees.*

We need to leave sufficient space for all of T's possible children and their descendants, i.e. $2 \cdot (2^{2^{i-1}} + 2^{2^{i-2}} + 2)$ layer $i-1$ fat nodes and their descendants. Thus T and its descendant fat nodes will need space $O(2^i \cdot 2^{2^i})$, and so the whole exponential tree uses space $O(N \log N)$.

Lemma 4. *The search time is $O(\log_B N)$.*

Proof. Consider layer i fat nodes, where $\sqrt{B} < 2^i \cdot 2^{2^i} \leq B$. Searching such a layer i fat node and all its descendant fat nodes takes $O(1)$ I/Os, since together they occupy $O(1)$ pages. Searching the higher portion of the tree, above layer i, takes $O(\log_B N + \log\log N - \log\log B)$ I/Os, for there are $O(\log\log N - \log\log B)$ layers in this portion of the exponential tree. But this is $O(\log_B N)$ I/Os in total. □

To reduce the space, each leaf is replaced by a record that stores between $\log N$ and $2\log N$ items in sorted order. This reduces the number of items in the exponential tree to $O(N/\log N)$ and hence the overall space used is reduced to $O(N)$.

Theorem 2. *There is a cache oblivious linear size data structure for an ordered set of N items supporting searches in worst case $O(\log_B N)$ I/Os and updates in amortized $O(\log_B N)$ I/Os.*

2.3 The Worst Case $O(\log_B N)$ Solution

The solution is quite similar to the solution with the same amortized complexity bound. The main changes to a layer i fat node tree T are the following:

- The implementation of the interface between fat nodes is slightly changed.
- The refreshing is performed incrementally over $\Theta(2^{2^{i-1}})$ insertions, and is initiated every $\Theta(2^{2^{i-1}})$ insertions in T.
- The splitting is performed incrementally over $\Theta(2^{2^i})$ insertions, and is initiated every $\Theta(2^{2^i})$ insertions in T.

The fat node interface is changed so that when a child fat node of T splits, T need not be immediately refreshed. In order to link to the new children generated by such a split, T is provided with space for an extra level of vertices: each link to a child fat node may be replaced by a vertex and, emanating from the vertex, links to the two new children fat nodes. However, only one additional layer of vertices will be allowed: we will ensure that T has been refreshed before any of the new child fat nodes splits.

A refreshing proceeds by copying all the leaf vertices in fat node T and then building internal vertices above them. Until a refreshing is complete searches and updates continue to be performed on the old version of T. We cross-link the copies of each leaf in the old and new versions of T so that any update resulting from a split of a child of T can be performed in both versions.

A split task proceeds by first splitting the top layer fat node, and then copying descendant fat nodes one by one in a depth first ordering. As a fat node is copied it is refreshed and split if need be. When the copying of a fat node is complete, searches will proceed through the new version(s) of the fat node(s). Consider the task that is splitting a layer j tree T, and suppose that it has just completed copying a descendant layer i fat node S. Now suppose that a search goes through S; it then follows a pointer to a layer $i-1$ fat node in the old version of T. Following this pointer could result in an I/O, even if S and its descendants would fit in a single block once their copying is complete. Call such a pointer a j-long pointer, and note that the span of a j-long pointer is contained within a layer $j+1$ tree. To guarantee a good search performance, we schedule tasks in such a way that if a search follows a k-long pointer, it will subsequently only follow j-long pointers for $j < k$. From this it follows that searches require only $O(\log_B N)$ I/Os.

We face the difficulty that, unfortunately, tasks could interfere with each other. We ensure that only one task at any time will be copying a fat node. Consider a split task $\mathcal{T}$ associated with fat node T. Suppose $\mathcal{T}$ is about to copy a descendant fat node S, but it finds S is already in the process of being copied by another task associated with a proper descendant of T; then $\mathcal{T}$ takes over the task, completes it, and finally does its own copying. This is readily implemented with a copy ID at the root of each fat node tree S identifying the name and layer of the task currently copying S, if any.

Interference also occurs if a layer h tree S is to be split but it is in the process of being copied by task $\mathcal{T}$ which is splitting a higher layer tree T. Then the task $\mathcal{S}$ for S will wait until $\mathcal{T}$ completes its copying of S and will then proceed with its own copying of S, unless taken over by some higher level task.

The handling of space is less obvious now. Each fat node is provided with double the space it needs; at any time it is stored in one half of the space and refreshing occurs in the other half. Also, lists of free space are needed within each layer tree to provide space for its children when they split. When the task $\mathcal{T}$ splitting a layer i tree finishes copying a layer $h < i$ subtree S and S's descendant subtrees, $\mathcal{T}$ releases the space for S, and if S had been in the process of splitting, the space for the old version of S.

Lemma 5. *A split takes $O(2^{2^i})$ time and I/Os.*

Next, we specify the choices of parameters. Now a layer i tree T has volume in the range $[2^{2^i}, 4 \cdot 2^{2^i})$. A split is initiated when its volume reaches $3 \cdot 2^{2^i}$ and will complete following at most 2^{2^i} insertions to T. The resulting trees have volumes in the range $[1.5 \cdot 2^{2^i} - 2 \cdot 2^{2^{i-1}}, 2.5 \cdot 2^{2^i} + 2 \cdot 2^{2^{i-1}}]$. It is readily checked that for $i \geq 2$, the trees created by the split have volumes in the range $[2^{2^i}, 3 \cdot 2^{2^i}]$.

Lemma 6. *The space used is $O(N \log^2 N)$.*

Proof. A layer i tree T has at most $4 \cdot 2^{2^i}/2^{2^{i-1}}$ child subtrees. □

To achieve space $O(N)$ we use buckets of size $\Theta(\log^2 N)$, implemented as two layers of records of size in the range $[\log N, 2\log N)$. Thus, we have shown:

Theorem 3. *There is a cache oblivious linear size data structure that supports searches and updates on an ordered set of items using $O(\log_B N)$ I/Os in the worst case.*

3 Planar Point Location

In this part of the extended abstract we only describe two results: both use preprocessing of $O(N \log_B N)$ I/Os and support queries using $O(\log_B N)$ I/Os; but the first result uses $O(N \log \log N)$ space, while the second uses only $O(N)$ space. We also briefly outline our stronger result which achieves linear space, $O((N/B) \log_{M/B} N)$ I/Os for preprocessing and $O(\log_B N)$ I/Os for queries, but requires the tall cache assumption.

We use the standard *trapezoidal decomposition*: vertical lines are extended up and down from each segment endpoint, terminating when they reach another segment. This partitions the plane into a collection of trapezoids, with vertical sides, and at most two non-vertical edges per trapezoid. A key tool in our preprocessing will be the *partition procedure*: given a parameter $m < N$, it selects $O(N/m)$ segments and forms the trapezoidal decomposition they define; each trapezoid will contain $\Theta(m)$ segments or part segments of the N input segments, and for each trapezoid these segments are identified. Such a trapezoidal decomposition is said to have *size* $\Theta(m)$.

The basic idea is to do a plane sweep with a vertical line from left to right, maintaining the collection of segments currently crossing its sweep line in sets of contiguous segments with sizes at least $m/7$ and at most m. It is helpful to set $m' = m/7$. Each pair of adjacent sets is separated by a segment already added to the set of selected segments. Each set is associated with a trapezoid T of the partition; T's left vertical boundary is known; its top and bottom boundaries are the selected segments bounding the set; its right boundary has yet to be specified.

We define the *current* size of a set to be the number of segments it contains (i.e. that cross the sweep line), and the *full* size to be the number of segments contained in the associated trapezoid. We ensure that the current and full sizes of a set are always in the range $[m', 7m']$. We further ensure that on creation, a set's sizes are in the range $[2m', 6m']$.

As left and right segment endpoints are swept over, the corresponding segments are respectively inserted into, or deleted from, the appropriate sets. When a set S's full or current size reaches an extreme value (due to the processing of an endpoint p, say) the vertical line through p provides the right boundary of the associated trapezoid. If S's current size lies outside $[2m', 6m']$, S is either split or merged with a neighbor set, as appropriate. A merge with set S' also results in the trapezoid associated with S' receiving the same right boundary. If the current size of the merged set is greater than $6m'$, the new set is immediately split. Following the splits and merges due to the processing of point p, for each remaining new set, a new trapezoid is instantiated, with left boundary the vertical line through p.

The splitting proceeds a little unusually and is explained next. The set of segments to be split all intersect the sweep line. Consequently they can be sorted along the sweep line. The middle m' segments are considered, and among these the one with the rightmost endpoint is chosen as the separator. The sets resulting from the split will thus have sizes in the range $[2.5m', 5m']$.

It is also possible that a point p swept over is the right endpoint of a selected segment σ. In this case, the two trapezoids bounded by σ are completed with right boundaries through p, and the corresponding sets are merged. If this new set has size more than $6m'$ it is split, and if has size more than $11m'$ it is partitioned by a three-way split, so as to obtain sets in the range $[3m', 6m']$ (two groups of m' segments each are considered, located in the one third and two thirds positions, rather than the middle).

The sweepline procedure works as follows. First, the segment endpoints are sorted according to their x values. Each endpoint is then considered in increasing x order. If it is a left endpoint, we locate the set containing this endpoint and add the incident segment to the set. Similarly, a right endpoint results in the deletion of the incident segment.

The locating is done by means of a search over the selected segments that intersect the sweep line. These segments are maintained in a dynamic search tree. Thus the search uses $O(\log_B m)$ I/Os and maintaining the search tree over the selected segments takes $O(m \log_B m)$ I/Os.

Lemma 7. *The partition procedure forms at most* $11N/m'$ *trapezoids.*

Proof. We count how many right vertical boundaries are created. There are a few ways for this to occur. First, a set can reach full size $7m'$; this requires at least m' insertions into the set since its creation. Second, a set can reach current size m'. This requires at least m' deletions from the set since its creation. Third, a set can be a neighbor of a set whose size drops to m'. Fourth, a set can be bounded by a selected segment whose right endpoint is reached.

There are at most N/m' sets in each of the first three categories. We show that there are at most $8N/m'$ sets in the fourth category by a credit argument. Each segment endpoint is given two credits. When processed, it gives its credits to the two selected segments bounding its set. Consider a selected segment σ whose right endpoint is reached. We show it has received at least m' credits. Consider the m' segments from which σ was selected. Let $\tau \neq \sigma$ be one of these segments. Suppose that when the sweepline reaches τ's right endpoint, τ is no longer in a set bounded by σ; then there must have been at least m' updates to the set containing τ that was created when σ was selected (so that τ could go to a different set). Each of these m' updates contributed one credit to σ. Otherwise, on reaching τ's right endpoint, the endpoint gives a credit to σ. Including the credit from σ's right endpoint, we see that σ always receives at least m' credits. □

Our first result applies the partition procedure with $m = \sqrt{N} \log N$. Within each trapezoid two sets of segments are identified: those fully contained in the trapezoid and those that touch or cross a vertical boundary (the only boundaries that can be crossed by segments). Each trapezoid and its fully contained segments are handled recursively. For each vertical boundary, we create a "side" search structure for the segments that cross it. Recall that we seek the segment immediately above a query point p.

The selected segments will be used to define the top layer search structure. Deeper layers are defined by the recursion on the trapezoids. The goal of the top search structure is two-fold: first to identify the segment among the selected segments which is directly above the query point; second, to identify the trapezoid defined by the selected segments containing the query point, and thereby enable a recursive continuation of the search. This search structure comprises two levels: the first level is a search tree over the x values given by the endpoints of the selected segments. For each pair of adjacent x values, the second level has a search structure on the selected segments spanning that pair of x values. Both of these can be implemented using binary search trees, laid out in a height-partitioned manner. Thus the space used in the top layer is $O(N/\log^2 N)$ and the search time within the top layer is $O(\log_B N)$.

The side search structure is simply a priority search tree. This can be implemented using a complete binary tree, which is then stored using the height-partitioning layout. In each layer, a query will search two side structures, one for each vertical boundary of the trapezoid being searched; each search will seek the first edge, if any, in the side structure on or above the query point. The search in layer i takes $O(\log_B 2^{2^i})$ I/Os, which summed over all layers is $O(\log_B N)$ I/Os. In order to avoid additive terms of $O(\log \log N)$ in the search time we allocate space for each subproblem recursively. Note that the above structure uses space $O(N \log \log N)$.

Theorem 4. *There is a cache oblivious data structure of size $O(N \log \log N)$ for planar point location that can be constructed using $O(N \log_B N)$ I/Os and supports queries using $O(\log_B N)$ I/Os.*

Now, we obtain a linear space solution. The basic idea is to create a bottom level comprising trapezoids holding $O(\log N)$ segments each. These segments can be stored contiguously, but in no particular order within each trapezoid; such a trapezoid can be searched with $O((\log N)/B)$ I/Os by means of a sequential scan. The bottom level is created by applying the partition procedure with $m = \log N$. Next, a planar point location search structure is build on the selected $O(N/\log N)$ segments, using our first solution. This structure is used to find the segment, among the selected segments, immediately above the query point. Finally, we have to connect the selected segment identified by the query to a particular trapezoid. The difficulty is that a single segment may bound multiple trapezoids. Thus, we need a third part to the data structure. Each segment is partitioned at the boundaries of the trapezoids it bounds, and a search tree is constructed over these boundary points. Having found a segment, the search continues by finding the segment portion immediately above the query point, and then proceeds to search the one trapezoid immediately beneath this segment portion.

Theorem 5. *There is a cache oblivious linear size data structure for planar point location that can be build using $O(N \log_B N)$ I/Os and supports queries using $O(\log_B N)$ I/Os.*

In our strongest result, we seek to build a trapezoidal decomposition of size $a \log N$ for some constant $a \geq 1$. In building the size $a \log N$ decomposition we also build a search structure that in $\Theta((N/B) \log_{M/B} N)$ I/Os can locate N query points, by returning the trapezoid containing the query point. More generally it can also locate $r < N$ query points using $\Theta((r/B) \log_{M/B} r + r\frac{\log(N/r)}{\log B})$ I/Os.

The basic approach is to build in turn trapezoidal decompositions of sizes $aN^{3/4}$, $aN^{(3/4)^2}, \ldots, a \log N$, using each decomposition as a stepping stone to the next one, a being a suitable constant. However, the most natural approach, which is to build each decomposition in turn appears to require re-reading of all the line segments to compute each decomposition, without being able to maintain locality at a sufficiently large granularity; this has an unacceptable cost of $\Omega((N/B) \log \log N)$ I/Os. A second approach, to proceed recursively, i.e. as a trapezoid in one decomposition is found, to immediately compute its partitioning in the smaller decomposition, appears to lead to segment duplication (of those segments that cross multiple trapezoids in a given decomposition, of which there can be many); this appears to entail unacceptable space (and work) $\Theta(N \cdot \text{polylog}\,(N))$.

Instead, our approach is to seek to identify individual trapezoids containing $\Theta(\log N)$ segments as early as possible, and then to remove them from the series of trapezoidal decompositions, putting them in the final size $a \log N$ decomposition right away. Thus our algorithm simultaneously builds the series of decreasing-size trapezoidal decompositions and the size $a \log N$ decomposition. Not surprisingly, some care is needed to choreograph this activity. Details can be found in [9].

Theorem 6. *There is a batched search structure for planar point location that uses space $O(N)$ and can be built with $O((N/B)\log_{M/B} N)$ I/Os and supports batched search of $r \leq N$ items using $O((r/B)\log_{M/B} r + r\log_B(N/r))$ I/Os.*

References

1. A. Aggarwal and J. S. Vitter. The I/O complexity of sorting and related problems. *Communications of the ACM* **31**, 1116–1127, 1988.
2. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. 25th VLDB Conference* (1999), pp 266–277.
3. A. Andersson. Faster Deterministic Sorting and Searching in Linear Space. In *Proc. 37th IEEE FOCS* (1996), pp. 135–141.
4. A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. 31st ACM STOC* (2000), pp. 335–342.
5. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, eds, *Handbook of Massive Data Sets*. Kluwer Academic, 2002.
6. L. Arge, D. E. Vengroff and J. S. Vitter. External-Memory Algorithms for Processing Line Segments in Geographic Information Systems (Extended Abstract). In *Proc. 6th European Symposium on Algorithms* (1995), LNCS 979, 295-310.
7. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, February 1972.
8. M. A. Bender, E. Demaine and M. Farach-Colton Cache-oblivous B-trees. In *Proc. 41st IEEE FOCS* (2000), pp. 399–409.
9. M. A. Bender, R. Cole and R. Raman. Exponential structures for efficient cache-oblivious algorithms. University of Leicester TR 2002/19, 2002.
10. M. A. Bender, Z. Duan, J. Iacono and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th ACM-SIAM SODA* (2002), pp. 29–38.
11. R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th ACM SPAA*, pp. 297–308, 1996.
12. G. S. Brodal, R. Fagerberg and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th ACM-SIAM SODA* (2002), pp. 39–48.
13. M. R. Brown, R. E. Tarjan. A data structure for representing sorted lists. *SIAM J. Comput.* **9** (1980), pp. 594–614.
14. M. T. Goodrich, J-J. Tsay, D. E. Vengroff and J. S. Vitter. External-Memory Computational Geometry (Preliminary Version). In *Proc. 34th IEEE FOCS* (1993), pp. 714–723.
15. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.
16. M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran Cache-oblivious algorithms. In *Proc. 40th IEEE FOCS* (1999), pp. 285–298.
17. K. Mehlhorn. *Data structures and algorithms, 1. Sorting and searching.* Springer, 1984.
18. H. Prokop. Cache-oblivious algorithms. MS Thesis, MIT, 1999.
19. N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. TR 00-02, King's College London, 2000. Prel. vers. in *Proc. ALENEX 2000*.

20. N. Rahman, R. Cole and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. 5th Workshop on Algorithm Engg.*, LNCS 2141, pp. 67–78, 2001.
21. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM* **29**, 1986, 669–679.
22. M. Thorup. Faster Deterministic Sorting and Priority Queues in Linear Space. *Proc. 9th ACM-SIAM SODA* (1998), pp. 550-555.
23. S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, Oct. 1997.
24. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys* **33** (2001) pp. 209–271.
25. D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.