Alcom-FT Technical Report Series
ALCOMFT-TR-02-52

Cache Oblivious Distribution Sweeping

Gerth Stølting Brodal[*,†] Rolf Fagerberg[*]

Abstract

We adapt the distribution sweeping method to the cache oblivious model. Distribution sweeping is the name used for a general approach for divide-and-conquer algorithms where the combination of solved subproblems can be viewed as a merging process of streams. We demonstrate by a series of algorithms for specific problems the feasibility of the method in a cache oblivious setting. The problems all come from computational geometry, and are: orthogonal line segment intersection reporting, the all nearest neighbors problem, the 3D maxima problem, computing the measure of a set of axis-parallel rectangles, computing the visibility of a set of line segments from a point, batched orthogonal range queries, and reporting pairwise intersections of axis-parallel rectangles. Our basic building block is a simplified version of the cache oblivious sorting algorithm Funnelsort of Frigo et al., which is of independent interest.

Keywords: Cache oblivious algorithms, sorting, distribution sweeping, computational geometry

1 Introduction

Modern computers contain a hierarchy of memory levels, with each level acting as a cache for the next. Typical components of the memory hierarchy are: registers, level 1 cache, level 2 cache, main memory, and disk. The time for accessing a level in the memory hierarchy increases from one cycle for registers and level 1 cache to figures around 10, 100, and 100,000 cycles

[*]BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: `{gerth,rolf}@brics.dk`. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[†]Supported by the Carlsberg Foundation (contract number ANS-0257/20).

for level 2 cache, main memory, and disk, respectively [16, p. 471], making the cost of a memory access depend highly on what is the current lowest memory level containing the element accessed. The evolution in CPU speed and memory access time indicates that these differences are likely to increase in the future [16, pp. 7 and 429].

As a consequence, the memory access pattern of an algorithm has become a key component in determining its running time in practice. Since classic asymptotic analysis of algorithms in the RAM model is unable to capture this, a number of more elaborate models for analysis have been proposed. The most widely used of these is the I/O model of Aggarwal and Vitter [1], which assumes a memory hierarchy containing two levels, the lower level having size M and the transfer between the two levels taking place in blocks of B elements. This model is illustrated in Figure 1. The cost of the computation in the I/O model is the number of blocks transferred. The model is adequate when the memory transfer between two levels of the memory hierarchy dominates the running time, which is often the case when the size of the data significantly exceeds the size of main memory, as the access time is very large for disks compared to the remaining levels of the memory hierarchy. By now, a large number of results for the I/O model exists—see e.g. the survey by Vitter [20]. A significant part of these results are for problems within computational geometry.

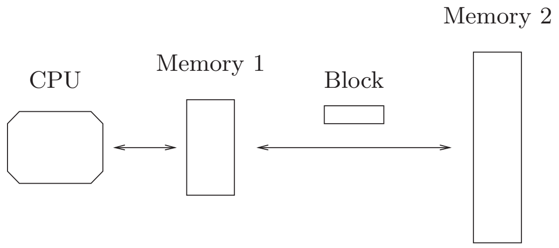

Figure 1: The I/O model

Recently, the concept of *cache oblivious* algorithms has been introduced by Frigo et al. [13]. In essence, this designates algorithms optimized in the I/O model, except that one optimizes to a block size B and a memory size M which are *unknown*. I/Os are assumed to be performed automatically by an off-line optimal cache replacement strategy. This seemingly simple change has significant consequences: since the analysis holds for any block and memory size, it holds for *all* levels of the memory hierarchy. In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized to each level automatically. Furthermore, the characteristics of the memory hierarchy do not need to be known, and do not need to be hardwired into the algorithm for the analysis to hold. This increases the portability of implementations of the algorithm, which is important in many situations, including production of software libraries and code delivered over the web. For further details on the concept of cache obliviousness, see [13].

Frigo et al. introduced the concept of cache oblivious algorithms and presented optimal cache oblivious algorithms for matrix transposition, FFT, and sorting [13]. Bender et al. [7], gave a proposal for cache oblivious search trees with search cost matching that of standard (cache aware) B-trees [5]. Simpler cache oblivious search trees with complexities matching that of [7] were presented in [8, 11]. Cache-oblivious data structures based on on exponential structures are presented in [6]. Recently, a cache-oblivious priority queue has been developed [3], which in turn gives rise to several cache-oblivious graph algorithms.

We consider cache oblivious algorithms within the field of computational geometry. Existing algorithms may have straightforward cache oblivious implementation—this is for example the case for the algorithm know as Graham's scan [15] for computing the convex hull of a point set [12, 18]. This algorithm first sorts the points, and then scans them while maintaining a stack containing the points on the convex hull of the points visited so far. Since the sorting step can be done by the Funnelsort algorithm of Frigo et al. [13] and a simple array is an efficient cache oblivious implementation of a stack, we immediately get a cache oblivious convex hull algorithm performing optimal $O(\mathrm{Sort}(N))$ I/Os, where $\mathrm{Sort}(N)$ is the optimal number of I/Os required for sorting. In this paper, we devise non-trivial cache oblivious algorithms for a number of problems within computational geometry.

In Section 2 we first present a version of the cache oblivious sorting algorithm Funnelsort of Frigo et al., which will be the basic component of our cache oblivious algorithms and which seems of independent interest due to its simplicity. In Section 3 we develop cache oblivious algorithms based on Lazy Funnelsort for a sequence of problems in computational geometry. Common to these problems is that there exist external memory algorithms for these problems based on the *distribution sweeping* approach of Goodrich et al. [14].

Goodrich et al. introduced distribution sweeping as a general approach for developing external memory algorithms for problems which in internal memory can be solved by a divide-and-conquer algorithm based on a plane sweep. Through a sequence of examples they demonstrated the validity of their approach. The examples mentioned in [14, Section 2] are: orthogonal line segment intersection reporting, the all nearest neighbors problem [21], the 3D maxima problem [17], computing the measure of a set of axis-parallel rectangles [9], computing the visibility of a set of line segments from a point [4], batched orthogonal range queries, and reporting pairwise intersections of axis-parallel rectangles.

We investigate if the distribution sweeping approach can be adapted to the cache oblivious model, and answer this in the affirmative by developing optimal cache oblivious algorithms for each of the above mentioned problems. Theorem 1 summarizes our results. These bounds are known to be optimal in the I/O model [14] and therefore are also optimal in the cache oblivious model.

Theorem 1 *In the cache oblivious model the 3D maxima problem on a set of points, computing the measure of a set of axis-parallel rectangles, the all nearest neighbors problem, and computing the visibility of a set of non-intersecting line segments from a point can be solved using optimal* $O(\mathrm{Sort}(N))$ *I/Os, and the orthogonal line segment intersection reporting problem, batched orthogonal range queries, and reporting pairwise intersections of axis-parallel rectangles can be solved using optimal* $O(\mathrm{Sort}(N) + \frac{T}{B})$ *I/Os, where* N *is the input size,* T *the output size, and* $\mathrm{Sort}(N)$ *the number of I/Os required to sort* N *elements.*

Goodrich et al. described distribution sweeping as a top-down approach. We instead describe it bottom-up, which facilitates our use of Funnelsort as a basic building block. The basic idea of the distribution sweeping approach is to sort the geometric objects, e.g. points and endpoints of line segments, w.r.t. one dimension and then apply a divide-and-conquer approach on this dimension where solutions to adjacent *strips* are merged to a solution for the union of the strips. This merging may be viewed as a sweep of the strips along another dimension. The details of the merging step is unique for each specific problem to be solved, but the overall structure of the method resembles Mergesort.

We note that the general method is not confined to problems within computational geometry—rather, any divide-and-conquer algorithm that combines solutions to subproblems in a merge-like fashion seems like a candidate for using the method, provided that the divide phase of the algorithm can be done as a separate preprocessing step by e.g. sorting. For such an algorithm, the applicability of the method in a cache oblivious setting is linked to the degree of locality of the information needed in each merge step, a point we elaborate on in the beginning of Sect. 3.

Arge [2] introduced the buffer tree—an external memory data structure which, among other uses, also can be used to solve the orthogonal line segment intersection problem, batched range queries, and reporting pairwise rectangle intersection. It is an open question, if there exists an efficient cache oblivious version of buffer trees or, more generally, how to adopt other external memory techniques and ideas to the cache oblivious model.

Preliminaries

By a *binary tree* we denote a rooted tree where nodes are either *internal* and have two children, or are *leaves* and have no children. The *size* $|T|$ of a tree T is its number of leaves. The *depth* $d(v)$ of a node v is the number of nodes (including v) on the path from v to the root. By *level* i in the tree we denote all nodes of depth i. We use $\log_x y$ as a shorthand for $\max\{1, \log_x y\}$.

2 Lazy Funnelsort

Frigo et al. in [13] gave an optimal cache oblivious sorting algorithm called *Funnelsort*, which may be seen as a cache oblivious version of Mergesort. In this section, we present a new version of the algorithm, termed *Lazy Funnelsort*. The benefit of the new version is twofold. First, its description, analysis, and implementation are, we feel, simpler than the original—features which are important for a problem as basic as sorting. Second, this simplicity facilitates the changes to the algorithm needed for our cache oblivious algorithms for problems in computational geometry. We also generalize Funnelsort slightly by introducing a parameter d which allows a trade-off between the constants in the time bound for Funnelsort and the strength of the "tall cache assumption" [13]. The choice $d = 3$ corresponds to the description in [13].

Central to Funnelsort is the concept of a *k-merger*, which for each invocation merges the next k^d elements from k sorted streams of elements. As a k-merger takes up space super-linear in k, it is not feasible to merge all N elements by an N-merger. Instead, Funnelsort recursively produces $N^{1/d}$ sorted streams of size $N^{1-1/d}$ and then merges these using an $N^{1/d}$-merger. In [13], a k-merger is defined recursively in terms of $k^{1/2}$-mergers and buffers, and the invocation of a k-merger involves a scheduling of its sub-mergers, driven by a check for fullness of all of its buffers at appropriate intervals.

Our modification lies in relaxing the requirement that all buffers of a merger should be checked (and, if necessary, filled) at the same time. Rather, a buffer is simply filled when it runs empty. This change allows us to "fold out" the recursive definition of a k-merger to a tree of binary mergers with buffers on the edges, and, more importantly, to define the merging algorithm in a k-merger directly in terms of nodes of this tree.

We define a k-merger as a perfectly balanced binary tree with k leaves. Each leaf contains a sorted input stream, and each internal node contains a standard binary merger. The output of the root is the output stream of the entire k-merger. Each edge between two internal nodes contains a buffer, which is the output stream of the merger in the lower node and is one of the two input streams of the merger in the upper node. The sizes of the buffers are defined recursively: Let $D_0 = \lceil \log(k)/2 \rceil$ denote the number of the middle level in the tree, let the *top tree* be the subtree consisting of all nodes of depth at most D_0, and let the subtrees rooted by nodes at depth D_0+1 be the *bottom trees*. The edges between nodes at depth D_0 and depth D_0+1 have associated buffers of size $\left\lceil k^{d/2} \right\rceil$, and the sizes of the remaining buffers is defined by recursion on the top tree and the bottom trees. For consistency, we think of the output stream of the root of the k-merger as a buffer of size k^d. In Figure 2, a 16-merger is illustrated.

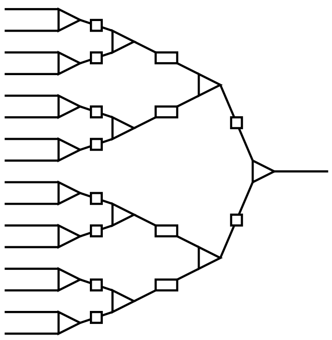

Figure 2: 16-merger

A k-merger, including the buffers associated with its middle edges, is laid out in memory in contiguous locations. This statement holds recursively for the top tree and for the bottom trees of the k-merger. In effect, a k-merger is the same as the van Emde Boas layout of a binary tree [19], except that edges now are buffers and take up more than constant space.

In Figure 3 our algorithm is shown for the binary merge process in each internal node of a k-merger. The last line means moving the smallest of

```
Procedure FILL(v)
    while v's output buffer is not full
        if left input buffer empty
            FILL(left child of v)
        if right input buffer empty
            FILL(right child of v)
        perform one merge step
```

Figure 3: The merging algorithm

the two elements in the fronts of the input buffers to the rear of the output buffer. The entire k-merger is simply invoked by a call FILL(r) on the root r of the merger. This will output k^d merged elements to the output buffer of the merger.

Concerning implementation details, we note that the input buffers of the merger may run empty during the merging. Exhausting of input elements should be propagated upward in the merger, marking a buffer as exhausted when both of its corresponding input buffers are exhausted. This is a simple extension of the code in Figure 3. We also note that buffers are emptied completely before they are filled, so they need not be implemented as circular arrays, in contrast to [13].

Lemma 1 *Let $d \geq 2$. The size of a k-merger (excluding its output buffer) is bounded by $c \cdot k^{(d+1)/2}$ for a constant $c \geq 1$. Assuming $B^{(d+1)/(d-1)} \leq M/2c$, a k-merger performs $O(\frac{k^d}{B}\log_M(k^d) + k)$ I/O's during an invocation.*

Proof. The space is given by the recursion formula $S(k) = k^{1/2} \cdot k^{d/2} + (k^{1/2} + 1) \cdot S(k^{1/2})$, which has a solution as stated.

For the I/O bound, we consider the recursive definition of buffer sizes in a k-merger, and follow the recursion until the space bound for the subtree (top tree or bottom tree) to recurse on is less than $M/2$, i.e. until $\bar{k}^{(d+1)/2} \leq M/2c$, where $\bar{k}$ is the number of leaves of the subtree. As $\bar{k}$ is the first such value, we know that $(\bar{k}^2)^{(d+1)/2} = \bar{k}^{d+1} > M/2c$. The buffers whose sizes will be determined during this partial recursion we denote *large* buffers. Removing the edges containing large buffers will partition the tree of the merger into a set of connected subtrees, which we denote *base trees.* By the tall cache assumption, a base tree and one block for each of the $\bar{k}$ buffers in its edges to leaves can be contained in memory, as $\bar{k} \cdot B \leq (M/2c)^{2/(d+1)} \cdot (M/2c)^{(d-1)/(d+1)} \leq M/2c$.

If the k-merger itself is a base tree, the merger and one block for each input stream will fit in memory, and the number of I/Os for outputting the k^d elements during an invocation is $O(k^d/B + k)$, as claimed. Otherwise, consider a call $\textsc{Fill}(v)$ to the root v of a base tree. This call will output $\Omega(\bar{k}^d)$ elements to the output buffer of v. Loading the base tree and one block for each of the $\bar{k}$ buffers just below the base tree into memory will incur $O(\bar{k}^{(d+1)/2}/B + \bar{k})$ I/Os. This is $O(1/B)$ I/Os per element output, since $\bar{k}^{d+1} > M/2c$ implies $\bar{k}^{d-1} > (M/2c)^{(d-1)/(d+1)} \geq B$ and hence $\bar{k} \leq \bar{k}^d/B$. During the call $\textsc{Fill}(v)$, the buffers just below the base tree may run empty, which will trigger calls to the nodes below these buffers. Such a call may evict the base tree from memory, leading to its reloading when the call finishes. However, a buffer of size $\Omega(\bar{k}^d)$ has been filled during this call, so the same calculation as above shows that the reloading of the base tree incurs $O(1/B)$ I/Os per element inserted into the buffer. The last time a buffer is filled, it may not be filled completely due to exhaustion. This happens only once for each buffer, so we can instead charge $O(1/B)$ I/Os to each position in the buffer in the argument above. As the large buffers are part of the space used by the entire k-merger, and as this space is sublinear in the output of the k-merger, this is $O(1/B)$ I/O per element merged.

In summary, charging an element $O(1/B)$ I/Os each time it is inserted into a large buffer will account for the I/Os performed. As $F = (M/2c)^{1/(d+1)}$ is the minimal number of leaves for a base tree, each element can be inserted in at most $\log_F k = O(d \log_M k) = O(\log_M k^d)$ large buffers, including the output buffer of the k-merger. From this the stated I/O bound follows. $\square$

Theorem 2 *Under the assumptions in Lemma 1, Lazy Funnelsort uses* $O(d\frac{N}{B}\log_M N)$ *I/Os to sort* N *elements.*

Proof. The algorithm recursively sorts $N^{1/d}$ segments of size $N^{1-1/d}$ of the input and then merges these using an $N^{1/d}$-merger. When the size of a segment in a recursive call gets below $M/2$, the blocks in this segment only needs to be loaded once into memory during the sorting of the segment, as the space consumption of a merger is linearly bounded in its output. For the k-mergers used at the remaining higher levels in the recursion tree, we have $k^d \geq M/2c \geq B^{(d+1)/(d-1)}$, which implies $k^{d-1} \geq B^{(d+1)/d} > B$ and hence $k^d/B > k$. By Lemma 1, the number of I/Os during a merge involving n' elements is $O(\log_M(n')/B)$ per element. Hence, the total number of I/Os per element is

$$O\left(\frac{1}{B}\left(1+\sum_{i=0}^{\infty}\log_M N^{(1-1/d)^i}\right)\right)=O\left(d\log_M(N)/B\right)\ .$$

□

3 Distribution Sweeping

Before going into the details for the various geometric problems, we below summarize the main technical differences between applying the distribution sweeping approach in the I/O model and in the cache oblivious model.

- In the I/O model, distribution sweeping uses $\Theta(M/B)$-ary merging. For cache oblivious algorithms, we do not know the parameters M and B, and instead use on binary merging. This is a simplification of the approach.
- In the I/O model, an entire merging process is completed before another merging process is started. In the cache oblivious model, we are building on (Lazy) Funnelsort, so this does not hold. Rather, a scheduling of the various merging processes takes place, and the intermediate outputs of merging processes are stored in buffers of limited size and used as input for other merging processes. This is a complication of the approach.

To illustrate the latter point, we note that in the distribution sweeping algorithms for batched orthogonal range queries, for orthogonal line segment intersection reporting, and for finding pairwise rectangle intersections, the merging process at a node needs to access the already merged part like a stack when generating the required output. In the I/O model this is not a problem, since there is always only one output stream present. In the cache oblivious model, the access to already merged parts is a fundamental obstacle, since this information may already have been removed by the merger at the parent node. Similar complications arise in the algorithm for all nearest neighbors. The solutions to these problems form a major part of the contribution of this paper.

On the other hand, for the 3D maxima problem and for computing the measure of a set of axis-parallel rectangles, this problem does not show up. The only difference from the merging performed in Lazy Funnelsort is that each input and output element is labeled with constant additional information, and that computing the labeling of an output element requires information of constant size to be maintained at the nodes of the merging process. For computing the visibility of a set of line segments from a point the situation is basically the same, except that some input points to a node in the merging process are removed during the merging.

3.1 3D Maxima Problem

Problem A d dimensional point $p = (p_1, \ldots, p_d)$ *dominates* a point $q = (q_1, \ldots, q_d)$ if and only if $p_i \geq q_i$ for all $1 \leq i \leq d$. The *maxima problem* consists of given a set of N points, to report the *maximal points* within the set, i.e. the points which are not dominated by any other input point.

In one dimension, the maxima problem is just the problem of computing the maximum of a set. To solve the two dimensional problem, the points are first sorted w.r.t. the first coordinate. The maximal points are then identified in a plane sweep through the points in decreasing order of the first coordinate. The first point is a maximal point, and the current point visited in the sweep is a maximal point if and only if it is not dominated by the last maximal point identified. During the sweep it suffices to store the second coordinate of the last maximal point identified.

The solution for the three dimensional maxima problem makes iterated use of the plane sweep part for the two dimensional maxima problem. First all points a sorted w.r.t. the third coordinate. A divide-and-conquer approach described in [4] is then applied on the third coordinate. The aim is to produce for each strip a stream of the points contained in the strip, with the points sorted w.r.t. decreasing first coordinate and with a point being marked if and only if it is a maximal point among the points in the strip.

The base case consist of strips containing a single point, which by definition must be marked. For a strip being the union of two strips A and B, with all points in A having smaller third coordinates than points in B, a merge is performed of the streams for A and B. While scanning B, the algorithm keeps track of the second coordinate y of the last point in B that is a maximal point in B when the third coordinate is ignored, using the above described algorithm for the two dimensional maxima problem. The algorithm picks the next point p from A and B with largest first coordinate. If p comes from B, then y is set to $\max\{y, p_2\}$. If p comes from A and is marked, then p is unmarked if and only if $p_2 \leq y$. Finally p is output. The correctness follows from the following facts: no point in A can dominate a point in B, a point $p \in B$ is a maximal point in $A \cup B$ if and only p is a maximal point in B, and a point $p \in A$ is a maximal point in $A \cup B$ if and only if p is a maximal point in A and is not dominated by any point in B.

For the initial sorting, we apply Lazy Funnelsort directly. For the subsequent divide-and-conquer approach, we note that the merge process described above is standard binary merging extended with the marking of points. We can therefore mimic Lazy Funnelsort by replacing the binary mergers in the nodes of a k-merger by the above described merge process. For each node we only need $O(1)$ additional space (for holding y), so the analysis of Lazy Funnelsort is still valid, i.e. the 3D maxima problem can be solved in $O(\text{Sort}(N))$ I/Os by a cache oblivious algorithm.

3.2 Measure of N Axis-Parallel Rectangles

Problem Given N axis-parallel rectangles in the plane, compute the measure of the union of the rectangles.

We first extract the $4N$ rectangle corners, and let each corner store the coordinates of the diagonally opposite corner. We sort the rectangle corners w.r.t. the first dimension, and then apply a divide-and-conquer approach on this dimension.

For a vertical strip S we will produce a stream containing the rectangle corners in S in increasing order w.r.t. the second dimension. Each corner p will be annotated with the one dimensional measure of the intersection of ℓ and (the union of) the rectangles with a corner in S, where ℓ is a horizontal line segment exactly covering the strip and lying strictly between p and the next point above p in the strip.

The base case consist of strips containing just two corner points, namely the two leftmost or the two rightmost corner points of a single rectangle. For a strip consisting of the union of two strips A and B, we merge the streams of points computed for the strips A and B by a plane sweep for increasing y. During the sweep we remember the last points a and b from A and B, and maintain a counter c_A describing how many rectangles with a corner in strip A spans strip B completely at the current y value, and similarly c_B describing how many rectangles with a corner in strip B spans strip A completely. These counts can easily be maintained in the merging process. For instance, if the next point is the lower left corner of a rectangle from strip A that spans strip B completely, we increase c_A, and if the next point is the upper right corner of a rectangle from B that spans strip A completely, we decrease c_B. When outputting a corner point p, we can compute the measure to be associated from the information maintained. As an example, if $p \in A$ and both strips A and B are spanned completely by rectangles at the sweep line, i.e. $c_A > 0$ and $c_B > 0$, then we assign the width of the strip

$A \cup B$ to p. If only A is spanned completely by a rectangle, i.e. $c_A = 0$ and $c_B > 0$, we assign to p the width of A plus the measure associated with b.

When the final merge has produced a strip containing all rectangles, the algorithm computes the total measure of the union of the rectangles by a linear scan through the computed one dimensional measures at the different y values, where each one dimensional measure is multiplied by the difference between the current y value and the y value of the next point, and the product is added the measure computed so far.

As for the 3D maxima problem, we perform exactly the binary merging as done by Funnelsort and only need $O(1)$ space at each node to store the status of the current sweep line, implying that the Funnelsort I/O bound also applies to this problem.

3.3 Visibility of Line Segments from a Point

Problem Given N non-intersecting line segments in the plane and a point p, compute the line segments visible from p.

We use a divide-and-conquer approach on the set of line segments, splitting sets arbitrarily into two subsets of equal size. For a given subset of the lines we will compute the sequence of line segments visible from p when rotating the viewing angle 360 degrees clockwise around p—more precisely, the sequence of endpoints of line segments visible together with the line segment visible to the right of the endpoint.

The base case consists of a single line, which will be represented twice in the sequence if it crosses the initial/final viewing angle. Given two sequences of visible endpoints for two sets of line segments A and B, the corresponding sequence for $A \cup B$ can be constructed by a rotation of the viewing angle through the visible endpoints of line segments from A and B. The merging process of the two sequences only has to remember the line segments from A and B which are visible at the current angle. Since a visible endpoint in B may not be visible in $A \cup B$, the output stream is only a subset of the endpoints visible for A and B.

Lazy Funnelsort does not require that input elements to a merger are also output by the merger, implying that the analysis for Lazy Funnelsort also applies to the visibility problem. Alternatively, we could output all endpoints of line segments, but marking the invisible points as such. Then the original Funnelsort algorithm and analysis would trivially apply.

3.4 Batched Orthogonal Range Queries

Problem Given N points in the plane and K axis-parallel rectangles, report for each rectangle R all points which are contained in R.

The basic distribution sweeping algorithm for range queries proceeds as follows. First all N points and the $2K$ upper left and upper right rectangle corners are sorted on the first coordinate. Each corner point contains a full description of the rectangle. After having sorted the points we use a divide-and-conquer approach on the first coordinate, where we merge the sequences of points from two adjacent strips A and B to the sequence of points in the strip $A \cup B$. All sequences are sorted on the second coordinate, and the merging is performed as a bottom-up sweep of the strip $A \cup B$. The property maintained is that if a rectangle corner is output for a strip, then we have reported all points in the strip that are contained in the rectangle.

While merging strips A and B, two lists L_A and L_B of points are generated: L_A (L_B) contains the input points from A (B), which are by now below the sweep line. If the next point p is an input point from A (B), we insert p into L_A (L_B) and output p. If p is a rectangle corner from A, and p is the upper left corner of a rectangle R that spans B completely in the first dimension, then the points in $L_B \cap R$ are reported by scanning L_B until the first point below the rectangle is found (if R only spans B partly, then the upper right corner of R is contained in B, i.e. $L_B \cap R$ has already been reported). On the RAM this immediately gives an $O(N \log N)$ time algorithm. The space usage is $O(N)$, since it is sufficient to store the L lists for the single merging process in progress. In the I/O model, a merging degree of $\Theta(\frac{M}{B})$ gives an $O(\mathrm{Sort}(N))$ time algorithm with a space usage of $O(\frac{N}{B})$ blocks.

Unfortunately, this approach does not immediately give an optimal cache oblivious algorithm. One problem is that the interleaved scheduling of the merge processes at nodes in a k-merger seems to force us to use $\Theta(n \log n)$ space for storing each input point in an L list at each level in the worst case. This space consumption is sub-optimal, and is also a problem in the proof of Theorem 2, where we for the case $N \leq M/2$ use that the space is linearly bounded.

We solve this problem in three phases: First we calculate for each node of a k-merger how many points will actually be reported against some query rectangle—without maintaining the L lists. By a simple change in the algorithm, we can then reduce the space needed at a node to be bounded by the reporting done at the node. Finally, we reduce the space consumption to $O(\frac{N}{B})$ blocks by changing the scheduling of the merging processes such that we force the entire merging process at certain nodes to complete before returning to the parent node.

In the following we consider a k-merger where the k input streams are available in k arrays holding a total of N points, and where $k = N^{1/d}$. In the first phase we do no reporting, but only compute how much reporting will happen at each of the $k-1$ nodes. We do so by considering a slightly different distribution sweeping algorithm. We now consider all N input points and all $4K$ corners of the rectangles. When merging the points from two strips A and B, we maintain the number a (b) of rectangles intersecting the current sweep line that span strip A (B) completely and have two corners in B (A). We also maintain the number of points r_A (r_B) in A (B) below the sweep line which cause at least one reporting at the node when applying the above algorithm. Whenever the next point is the lower left (right) corner of a rectangle spanning B (A) completely, b (a) is increased. Similarly we decrease the counter when a corresponding topmost corner is the next point. If the next point is an input point from A (B), we increase r_A (r_B) by one if and only if a (b) is nonzero. Since the information needed at each node is constant, we can apply the Lazy Funnelsort scheduling and the analysis from Lemma 1 for this first phase.

By including the lower rectangle corner points in the basic reporting algorithm, we can simultaneously with inserting points into L_A and L_B keep track of a and b, and avoid inserting a point from A (B) into L_A (B) if the point will not be reported, i.e. if a (b) is zero. This implies that all points inserted into L_A and L_B will be reported at least once, so the space $O(r_A + r_B)$ required for L_a and L_b is bounded by the amount of reporting generated at the node.

Finally, to achieve space linear in the total input N of the k-merger (not counting the space needed for storing the reporting generated), we will avoid allocating the L lists for all nodes simultaneously if this will require more than linear space. The reporting generated by a k-merger will be partitioned into iterations, each of which (except the last) will generate $\Omega(N)$ reporting using space $O(N)$. The details are as follows. First we apply the above algorithm for computing the r_A and r_B values of each node of the k-merger. In each iteration we identify (using a post-order traversal principle) a node v in the k-merger where the sum of the r_A and r_B values at the descendants is at least N, and at most $3N$ (note: for each node we have $r_A + r_B \leq N$). If no such node exists, we let v be the root. We first allocate an array of size $3N$ to hold all the L_A and L_B lists for the descendants of v. We now complete the entire merging process at node v, by repeatedly applying Fill(v) until the input buffers of v are exhausted. We move the content of the output buffer of v to a temporary array of size N, and when the merging at v finished we move the output to a global array of size N which holds the final merged lists of several nodes simultaneously. If the k input streams have size $N_1, \ldots, N_k$, and node v spans streams $i..j$, the merged output of v is stored at positions $1 + \sum_{\ell=1}^{i-1} N_i$ and onward. When the merging of v is finished, we set r_A and r_B of all descendants of v to zero.

For the analysis, we follow the proof of Lemma 1. We first note that by construction, we use space $\Theta(N)$ and in each iteration (except the last) generate $\Omega(N)$ reporting. If $N \leq M/2c$, all computation will be done in internal memory, when the input streams first have been loaded into memory, i.e. the number of I/Os used is $O(\frac{N}{B} + \frac{T}{B})$. For the case $N > M/2c$, i.e. $k < \frac{N}{B}$, we observe that each base tree invoked only needs to store $O(1)$ blocks from the head of each L list in the nodes of the base tree. Writing a point to an L list can then be charged to the later reporting of the point. Reading the first blocks of the L lists in a base tree has the same cost as reading the first blocks of each of the input streams to the base tree. We conclude that the I/Os needed to handle the L lists can either be charged to the reporting or to the reading of the input streams of a base tree. The total number of I/Os used in an iteration is $O(k + \frac{N}{B} + \frac{T'}{B})$, where T' is the amount of reporting, plus the number of I/Os used to move points from a base tree to the base next. Over all iterations, the latter number of I/Os is at most $O(\frac{N}{B} \log_M N)$. We conclude that the k-merger in total uses $O(\frac{N}{B} \log_M N + \frac{T}{B})$ I/Os and uses $O(\frac{N}{B})$ blocks of space. Analogous to the proof of Theorem 2 it follows that the entire algorithm uses $O(d\frac{N}{B} \log_M N + \frac{T}{B})$ I/Os.

3.5 Orthogonal Line Segment Intersection Reporting

Problem Given N axis-parallel line segments in the plane, report all intersection points between horizontal and vertical line segments.

The algorithm is the same as for the batched range query problem, except for the following modifications. The analysis remains the same. The points at the leaves are the $2N$ endpoints of the N line segments. While merging the points in two strips A and B, the list L_A (L_B) contains the horizontal line segments that completely span the strip A (B) and have an endpoint in B (A). If the next point is the topmost endpoint of a vertical line segments in A (B), then all horizontal segments in L_A (L_B) are reported, until a horizontal line segment below the vertical line segment is reached.

3.6 Pairwise Rectangle Intersections

Problem Given N axis-parallel rectangles in the plane, report all pairwise intersections between the rectangles.

Since the pairwise rectangle intersection problem can be reduced to solving one orthogonal segment intersection reporting problem (reporting intersections of sides of rectangles) and one batched range searching problem (reporting rectangle corners within rectangles), as noted in [2, 10], the I/O bound in Theorem 1 follows immediately.

3.7 All Nearest Neighbors

Problem Given N points in the plane, compute for each point which other point is the closest.

We solve the problem in two phases. After the first phase, each point p will be annotated by another point p_1 which is at least as close to p as the closest among all points lying *below* p. The point p_1 itself does not need to lie below p. If no points exist below p, the annotation may be empty. The second phase is symmetric, with *above* substituted for *below*, and will not be described further. The final result for a point p is the closest of p_1 and the corresponding annotation from the second phase.

In the first phase, we sort the points on the first dimension and apply a divide-and-conquer approach from [21] on this dimension. For each vertical strip S, we will produce a stream containing the points in S in decreasing order w.r.t. the second dimension, with each point annotated by *some* other point p_1 from S (or having empty annotation). The divide-and-conquer approach will be patterned after Lazy Funnelsort, and for streams analogous to output streams of k-mergers, the annotation will fulfill an invariant as above, namely that p_1 is at least as close as the closest among the points from S lying below p (for streams internal to k-mergers, this invariant will not hold).

The base case is a strip containing a single point with empty annotation. For a strip being the union of two strips A and B, we merge the streams for A and B by a downward plane sweep, during which we maintain two *active sets* S_A and S_B of copies of points from A and B, respectively. For clarity, we in the discussion below refer to such a copy as an *element* x, and reserve the term *point* p for the original points in the streams being merged.

These active sets are updated each time the sweepline passes a point p. The maintenance of the sets are based on the following definition: Let c

denote the intersection of the horizontal sweepline and the vertical line separating A and B, let p be a point from S, let p_1 be the point with which p is annotated, and let d denote Euclidean distance. By $U(p)$ we denote the condition $d(p, p_1) \leq d(p, c)$, where $d(p, p_1)$ is taken as infinity if p has empty annotation. If $U(p)$ holds and p is in A (B), then no point in B (A) lying below the sweepline can be closer to p than p_1.

We now describe how the active sets are updated when the sweepline passes a point $p \in A$. The case $p \in B$ is symmetric. We first calculate the distance $d(p, x)$ for all elements x in $S_A \cup S_B$. If this is smaller than the distance of the current annotation of x (or p, or both), we update the annotation of x (or p, or both). A copy of the point p is now added to S_A if condition $U(p)$ does not hold. In all cases, p is inserted in the output stream of the merge process. Finally, if for any x in $S_A \cup S_B$ condition $U(x)$ is now true, we remove x from its active set. When the sweepline passes the last point of S, we remove any remaining elements in S_A and S_B.

By induction on the number of points passed by the sweepline, all elements of S_A are annotated by a point at least as close as any other element currently in S_A. Also, $U(x)$ is false for all $x \in S_A$. As observed in [21], this implies that for any two elements x_1 and x_2 from S_A, the longest side of the triangle $\triangle x_1 c x_2$ is the side $x_1 x_2$, so by the law of cosines, the angle $\angle x_1 c x_2$ is at least $\pi/3$. Therefore S_A can contain at most two elements, since the existence of three elements x_1, x_2, and x_3 would imply an angle $\angle x_i c x_j$ of at least $2\pi/3$ between two of these. By the same argument we also have $|S_B| \leq 2$ at all times.

Let $I(p, X)$ denote the condition that the annotation of p is a point at least as close as the closest point among the points lying below p in the strip X. Clearly, if a point $p \in A$ is passed by the sweepline without having a copy inserted into S_A, we know that $I(p, B)$ holds. If a copy x of p is inserted into S_A, it follows by induction on the number of points passed by the sweepline that $I(x, B)$ holds when x is removed from S_A. Similar statements with A and B interchanged also hold.

As said, our divide-and-conquer algorithm for phase one is analogous to Lazy Funnelsort, except that the merge process in a binary node of a k-merger will be the sweep line process described above. We allocate $O(1)$ extra space at each node to store the at most four copies contained in the two active sets of the merge process. We will maintain the following invariant: when a k-merger spanning a strip S finishes its merge process, condition $I(p, S)$ holds for all points p in output stream of the k-merger. Correctness of the algorithm follows immediately from this invariant. From the statements in the previous paragraph we see that the invariant is maintained if we ensure that when a k-merger finishes, the annotations of points p in its output stream have been updated to be at least as close as the annotations of *all* copies of p removed from active sets during the invocation of the k-merger.

To ensure this, we keep the copies of a point p in a doubly linked list along the path toward the root of the k-merger. The list contains all copies currently contained in active sets, and has p itself as the last element. Note that in a k-merger, the merge processes at nodes are interleaved—part of the output of one process is used as input for another before the first process has finished—so the length of this list can in the worst case be the height of the k-merger.

Consider a merge step in a node v in a k-merger which moves a point p from an input buffer of v to its output buffer. During the step, several types of updates of the linked list may be needed:

1. If p is currently contained in a list, the forward pointer of the next-to-last element needs to be updated to p's new position.

2. A copy x of p may be inserted in an active set of v. If p is currently not in a list, a new list containing x and p is made. Otherwise, x is inserted before p in p's list.

3. Elements of active sets of v may be removed from the sets. Each such element x should be deleted from its linked list.

This updating is part of the algorithm. Additionally, in the third case the annotation of x is propagated to the next element y in the list, i.e. the annotation of y is set to the closest of the annotations of x and y. This ensures the invariant discussed above.

We now analyze the I/O complexity of the algorithm. As the space usage of a node in a k-merger is still $O(1)$, the analysis of Lazy Funnelsort is still valid, except that we need to account for the I/Os incurred during updates of the linked lists.

Whenever a base tree not residing in memory is touched by the algorithm, we in the proof of Lemma 1 charged the cost of loading the entire base tree into memory. This implies that the cost of updating pointers starting and ending in the same base tree has already been accounted for.

We also charged the cost of loading $O(\bar{k})$ blocks memory, where $\bar{k}$ is the number of leaves in the base tree, and proved that these blocks can reside in memory concurrently with the base tree. Hence, we may assume that the memory always contains the blocks for the memory locations pointed to by the $O(\bar{k})$ forward pointers of elements in active sets of the nodes of the base tree. This is because the initial set of such blocks for a newly loaded base tree has already been accounted for, and new such blocks only appear when a new element x is inserted into the active sets of a node. When x, being a copy of point p, is inserted into an active set, the forward pointer of x should point to p, which at the same time is inserted into the output buffer of the node. Hence, loading the block pointed to by the forward pointer of x has already been done by the output process of the node. We conclude that the cost of updating the reverse of these pointers, i.e. the backwards pointers pointing to elements in active sets of the current node, already has been accounted for in the proof of Lemma 1.

What remains is to bound the I/O cost of updating forward pointers originating outside the base tree of the current node and ending at the current input element or at an active set of the current node. We will prove that amortized over the entire invocation of the k-merger, there are sufficiently few such pointers to be updated for us to afford the $O(1)$ I/Os incurred during an update. Our amortization argument uses potential functions on these pointers to pay for the I/Os incurred during updates.

For a forward pointer ℓ, let its *maximal buffer* $m(\ell)$ be the largest buffer on the path in the k-merger from its starting point to its endpoint (including the buffer containing the endpoint if ℓ is the last pointer of the list), and denote ℓ a *large pointer* if $m(\ell)$ is a large buffer according to the definition in the proof of Lemma 1. The buffer $m(\ell)$ resides on the middle level of the smallest tree $T(\ell)$ in the recursive definition of buffer sizes that contains both starting point and endpoint of ℓ. Let $k(\ell)$ be the size of the top tree of $T(\ell)$. We view the $\log k(\ell)$ levels of the top tree of $T(\ell)$ as composed of $2\log k(\ell) - 1$ semi-levels, each containing all nodes or all edges at the same depth in $T(\ell)$. The crucial observation is that each time a forward pointer is updated, its new endpoint is at least one semi-level higher than the old. Therefore we to each large forward pointer ℓ assign a potential of $2\log k(\ell) - i$, where i is the number of semi-levels between $m(\ell)$ and the endpoint of ℓ.

During the lifetime of a forward pointer ℓ, the size of $m(\ell)$ can only increase, since the starting point of ℓ is fixed and the endpoint can only move upward in the k-merger. For all updates of large forward pointers where $m(\ell)$ and hence $k(\ell)$ do not change, the decrease in potential will be at least one, which will cover the $O(1)$ I/Os for the update. We now analyze the remaining cases.

One case is an update of type 3 above, where $m(\ell')$ for the forward pointer ℓ' of the removed element x is larger than $m(\ell)$ for the forward pointer ℓ of the predecessor of x in the list. During the update, ℓ' vanishes. The hereby released potential is exactly the required new potential of ℓ, and the cost of the update is covered by the old potential of ℓ, as this is at least one.

The other case is an update of type 1, where p is inserted into a buffer m larger than the current $m(\ell)$ for the forward pointer ℓ pointing to p. In this case we use the last unit of the potential of ℓ to cover the update. However, we must provide a new potential for ℓ of $2\log k$, where k is the new value of $k(\ell)$. By the nature of the procedure FILL(), all of buffer m will be filled before any point is removed from it, so we can postpone providing the potential for such pointers until FILL() is finished filling buffer m. At that point in time, there are few such pointers compared to the size of m: they all have starting points within the bottom tree of the tree of which m is a middle buffer, and this bottom tree contains $O(k)$ pointers, while m contains $\Theta(k^d)$ points. We can therefore provide the total potential required by charging $\Theta(k \log k/k^d)$ per point inserted into m. If this is $O(1/B)$, we do not charge more than what in the proof of Lemma 1 is already charged for inserting an element into a large buffer. Such a statement can be proved under a slightly stronger tall cache assumption than the one in Lemma 1. For instance, assuming $B^{(d+1/(d-1-\delta))} \leq M$ for $\delta > 0$, we can prove $k \log k/k^d \leq 1/(\delta B)$: For all $x > 0$ we have $\log(x) \leq x$, which implies $\log(k) \leq k^\delta/\delta$. As m is a large buffer, we know from the proof of Lemma 1 that $k \geq M^{1/(d+1)}$. All in all we have $k \log k/k^d \leq 1/\delta \cdot 1/k^{d-1-\delta} \leq 1/\delta \cdot 1/M^{(d-1-\delta)/(d+1)} \leq 1/(\delta B)$.

Finally, when a pointer becomes large for the first time (during a type 1 or type 2 update), the actual I/O for the update has previously been accounted for, but the amortization argument now requires us to provide new potential. The argument for the second case above shows that this can be covered by charging $O(1/B)$ to each element inserted in the buffer currently being filled. This concludes the analysis of the algorithm.

Acknowledgment

We would like to thank Riko Jacob and Sven Skyum for encouraging discussions.

References

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

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer Verlag, Berlin, 1995.

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing*. ACM Press, 2002. To appear.

[4] M. J. Atallah and J.-J. Tsay. On the parallel-decomposability of geometric problems. *Algorithmica*, 8:209–231, 1992.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[6] M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2002. These proceedings.

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

[8] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 29–39, 2002.

[9] J. L. Bentley. Algorithms for Klee's rectangle problems. Carnegie-Mellon University, Pittsburgh, Penn., Department of Computer Science, unpublished notes, 1977.

[10] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29:571–577, 1980.

[11] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, Berlin, 1997.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[14] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Ann. Symp. on Foundations of Computer Science*, pages 714–723, 1993.

[15] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1:132–133, 1972.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[17] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectores. *Journal of the ACM*, 22(4):469–476, Oct. 1975.

[18] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, Berlin, 1985.

[19] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[20] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[21] D. E. Willard and Y. C. Wee. Quasi-valid range querying and its implications for nearest neighbor problems. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 34–43. ACM Press, 1988.