I/O-Efficient Well-Separated Pair Decomposition and Applications*

Sathish Govindarajan† Tamás Lukovszki‡ Anil Maheshwari§
Norbert Zeh¶

August 14, 2005

Abstract

We present an external-memory algorithm to compute a well-separated pair decomposition (WSPD) of a given point set S in $\mathbb{R}^d$ in $\mathcal{O}(\text{sort}(N))$ I/Os, where N is the number of points in S and $\text{sort}(N)$ denotes the I/O-complexity of sorting N items. (Throughout this paper, we assume that the dimension d is fixed). As applications of the WSPD, we show how to compute a linear-size t-spanner for S within the same I/O-bound and how to solve the K-nearest-neighbour and K-closest-pair problems in $\mathcal{O}(\text{sort}(KN))$ and $\mathcal{O}(\text{sort}(N+K))$ I/Os, respectively.

Keywords: External-memory algorithms, computational geometry, well-separated pair decomposition, spanners, closest-pair problem, proximity problems.

1 Introduction

Many geometric applications require computations that involve the set of all distinct pairs of points (and their distances) in a set S of N points in d-dimensional Euclidean space. One such problem is computing, for every point in S, its nearest neighbour in S—the all-nearest-neighbour problem. Voronoi diagrams and multi-dimensional divide-and-conquer are the traditional techniques used for solving several distance-based geometric problems, especially

*Research supported by NSERC. A preliminary version appeared in [28].

†Department of Computer Science, Duke University, Durham, NC 27708-0129, USA. gsat@cs.duke.edu.

‡Heinz-Nixdorf-Institut and Department of Computer Science, University of Paderborn, Germany. tamas@hni.upd.de. This research was done while visiting Carleton University. Research partially supported by DFG-grant SFB376 and NSERC.

§School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada. maheshwa@scs.carleton.ca.

¶Faculty of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, NS B3H 1W5, Canada. nzeh@cs.dal.ca.

in two and three dimensions. Callahan and Kosaraju [15] introduced the well-separated pair decomposition (WSPD) as a data structure to cope with higher-dimensional geometric problems. It consists of a binary tree T and a list of "well-separated" pairs of subsets of S. The leaves of T represent the points in S; every internal node represents the subset of points corresponding to its descendent leaves. For every well-separated pair $\{A, B\}$, both A and B are sets represented by nodes in T. Intuitively, a pair $\{A, B\}$ is well-separated if the distance between A and B is significantly greater than the distance between any two points within A or B. It turns out that, for many problems, it is sufficient to perform only a constant number of operations on every pair $\{A, B\}$ instead of performing $|A||B|$ operations on the corresponding pairs of points. Moreover, for fixed d, a WSPD of $\mathcal{O}(N)$ pairs of subsets can be constructed in $\mathcal{O}(N \log N)$ time. This results in fast sequential, parallel, and dynamic algorithms for a number of problems on point sets (e.g., see [5, 10, 11, 13, 14, 15]). Here we extend these results to external memory.

1.1 Previous Work

1.1.1 The I/O-Model

In the I/O-model [1], a computer is equipped with a two-level memory consisting of *internal memory* (RAM) and *external memory* (disk). The internal memory is assumed to be capable of holding M data items. The disk is divided into blocks of B consecutive data items. All computation has to happen in internal memory. Data is transferred between internal memory and disk by means of *I/O-operations* (I/Os); each such operation transfers one block of data between internal memory and disk. The complexity of an algorithm in the I/O-model is the number of I/Os it performs. Surveys of relevant work in the external-memory setting are given in papers by Arge [2] and Vitter [48] and in [33]. It has been shown that sorting an array of size N takes $\text{sort}(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os [1]. Scanning an array of size N takes $\text{scan}(N) = \Theta(N/B)$ I/Os. Since the scanning bound can be seen as the equivalent of the linear time bound required to perform the same operation in internal memory, we refer to $\mathcal{O}(N/B)$ I/Os as a *linear number of I/Os*.

1.1.2 Spanners and Proximity Problems

Proximity problems and the construction of geometric spanners—problems whose solutions are among the applications of the WSPD—have a rich history. Before the study of the I/O-complexities of these problems, much research has focused on the development of efficient internal-memory algorithms for these problems. This work is reviewed in [24, 34, 46]. We recall the most important internal-memory results in this area and then discuss the state of the art in external memory. For a comprehensive discussion of the WSPD and its applications, we refer the reader to [11].

Closest pair. The *closest-pair problem* is that of finding the distance between the closest two points in a point set in $\mathbb{R}^d$. A trivial $\mathcal{O}(N^2)$ time solution to this problem is to

examine all possible point pairs and report the shortest distance. But this leaves a gap to the $\Omega(N \log N)$ lower bound provable in the algebraic computation tree model [7]. The first optimal algorithms solving this problem in two dimensions are due to Shamos [43] and Shamos and Hoey [44]. Lenhof and Smid [31] present a simple and practical algorithm that solves the problem in $\mathcal{O}(N \log N)$ time. Their algorithm uses the floor function and indirect addressing and, thus, does not conform with the algebraic model of computation. Rabin [38], Khuller and Matias [30], and Seidel [42] present randomized closest-pair algorithms that take expected linear time. The algorithm of [42] takes $\mathcal{O}\left(N \frac{\log N}{\log \log N}\right)$ time with high probability.

Dynamic closest pair. Data structures for maintaining the closest pair in a point set that changes dynamically under insertions and deletions have been discussed in [9, 12, 14, 26, 41], culminating in the optimal structure of [9], which uses linear space and maintains the closest pair under point insertions and deletions in $\mathcal{O}(\log N)$ time. This structure dynamically maintains a fair split tree (see Section 2) of the point set and extracts the new closest pair from the updated tree after every update.

***K* closest pairs.** An obvious extension of the closest-pair problem is that of reporting the K *closest pairs* instead of only the closest pair. For this problem, the first two algorithms, due to Smid [45], take $\mathcal{O}\left(N \log N + N\sqrt{K} \log K\right)$ time in two dimensions and $\mathcal{O}\left(N^{4/3} \log N + N\sqrt{K} \log K\right)$ time in d dimensions, where $d > 2$. Dickerson, Drysdale, and Sack [22] propose a simple $\mathcal{O}(N \log N + K \log K)$ time algorithm for the planar case. In [23], Dickerson and Eppstein extend the result of [22] to higher dimensions, achieving the same running time as the algorithm of [22] for the planar case. If the distances do not need to be reported in sorted order, improved $\mathcal{O}(N \log N + K)$ time algorithms are presented in [23, 40]. Lenhof and Smid [31] give a much simpler algorithm achieving the same running time; but they use indirect addressing, thereby leaving the algebraic model of computation. An $\mathcal{O}(N \log N + K)$ time K-closest-pair algorithm based on the WSPD is presented in [11].

All nearest neighbours. Another extension of the closest-pair problem is that of computing the *nearest neighbour* for every point in the set. This problem is also known as the *all-nearest-neighbour problem*. The algorithms of [44] can easily be extended to compute all nearest neighbours in two dimensions in optimal $\mathcal{O}(N \log N)$ time. Bentley [8] shows how to extend his closest-pair algorithm for the d-dimensional case so that it can solve the all-nearest-neighbour problem in $\mathcal{O}(N \log^{d-1} N)$ time. The first $\mathcal{O}(N \log N)$ time algorithm to solve the all-nearest-neighbour problem in higher dimensions is due to Clarkson [18]. The algorithm is randomized; hence, the running time is expected. Vaidya [47] gives the first deterministic $\mathcal{O}(N \log N)$ time algorithm for this problem. Callahan and Kosaraju [15] show that the WSPD can also be used to solve the all-nearest-neighbour problem in $\mathcal{O}(N \log N)$ time. In fact, they show that the more general problem of computing the K nearest neighbours for every point in the point set can be solved in $\mathcal{O}(N \log N + KN)$ time.

Spanners. The concept of spanner graphs was introduced by Chew [16]. A geometric t-*spanner* is a subgraph of the complete Euclidean graph of a point set that has $\mathcal{O}(N)$ edges and approximates inter-point distances to within a multiplicative factor of $t > 1$, called the *spanning ratio* of the graph. Keil and Gutwin [29] prove that the spanning ratio of the Delaunay triangulation is no more than $\frac{2\pi}{3\cos(\pi/6)} \approx 2.42$. Unfortunately, by a result of [16], the Delaunay triangulation cannot be used when a spanning ratio arbitrarily close to 1 is desired. In fact, it is easy to show that there are point sets such that no planar graph over such a point set has spanning ratio less than $\sqrt{2}$ in the worst case. The first to show how to construct a t-spanner in the plane, for t arbitrarily close to one, were Keil and Gutwin [29]. Independently, Clarkson [19] discovered the same construction for $d = 2$ and $d = 3$. Ruppert and Seidel [39] generalize the result to higher dimensions, using the construction of a θ-frame due to Yao [49]; the algorithm takes $\mathcal{O}\left(N \log^{d-1} N\right)$ time. Arya, Mount, and Smid [5] combine the θ-graph of Ruppert and Seidel with skip lists [37] to obtain a t-spanner of spanner diameter $\mathcal{O}(\log N)$, with high probability; that is, for any two points p and q, there exists a path consisting of $\mathcal{O}(\log N)$ edges whose length is within a factor of t of the Euclidean distance between p and q. The WSPD [15] can be used to obtain an $\mathcal{O}(N \log N)$ time algorithm for constructing a t-spanner [13]. In [5], it is shown that the spanner obtained in this way can be constructed more carefully to guarantee that the spanner diameter is $\mathcal{O}(\log N)$.

Results in external memory. A number of external-memory algorithms for proximity problems and problems related to spanner construction have been proposed in the last decade. See [2, 48] for surveys. Randomized external-memory algorithms for constructing Voronoi diagrams in two and three dimensions have been proposed in [21]. Using these algorithms, one can compute the Delaunay triangulation of a point set in $\mathbb{R}^2$ and scan its edge list to identify the closest pair. This takes $\mathcal{O}(\text{sort}(N))$ I/Os. External-memory algorithms for finding all nearest neighbours for a set of N points in the plane and for other geometric problems in the plane are discussed in [27]; the all-nearest-neighbour algorithm of [27] takes $\mathcal{O}(\text{sort}(N))$ I/Os. An I/O-efficient data structure for dynamically maintaining the closest pair in a point set is described in [12]. The data structure can be updated in $\mathcal{O}(\log_B N)$ I/Os per point insertion or deletion. For $\Omega(\text{sort}(N))$ I/O lower bounds for computational geometry problems in external memory, including the closest-pair problem, see [4]. We are not aware of any previous results on solving these problems I/O-efficiently in higher dimensions, nor of any results relating to the I/O-efficient construction of spanner graphs, except for the algorithm of [21] for constructing the Delaunay triangulation.

1.2 New Results

In this paper, we present external-memory algorithms for a number of proximity problems on a point set S in d-dimensional Euclidean space. The basic tool is an algorithm that constructs the WSPD of S in $\mathcal{O}(\text{sort}(N))$ I/Os. Given the WSPD, we show that the K-nearest-neighbour and K-closest-pair problems can be solved in $\mathcal{O}(\text{sort}(KN))$ and $\mathcal{O}(\text{sort}(N + K))$

I/Os, respectively. We also argue that a t-spanner of spanner diameter $\mathcal{O}(\log N)$ can be derived from the WSPD in $\mathcal{O}(\text{sort}(N))$ I/Os.

All our algorithms follow the strategies of algorithms by Callahan and Kosaraju for these problems [10, 13, 15]. However, the original algorithms are not I/O-efficient or, in the case of the parallel algorithms for computing the WSPD [10] and for computing nearest neighbours [10], lead to suboptimal results when translated into I/O-efficient algorithms using the PRAM-simulation technique of [17]. Our contribution is to provide non-trivial I/O-efficient implementations of the high-level steps in the algorithms by Callahan and Kosaraju, which lead to the I/O-complexities stated above.

To the best of our knowledge, our results are the first I/O-efficient algorithms obtained for problems in higher-dimensional computational geometry. No I/O-efficient algorithms for computing a fair split tree or a WSPD were known. For the K-nearest-neighbour and K-closest-pair problems, optimal algorithms were presented in [27] for the case where $d = 2$ and $K = 1$. In [4], it is shown that computing the closest pair of a point set requires $\Omega(\text{sort}(N))$ I/Os, which implies the same lower bound for the more general problems we consider in this paper. Other I/O-efficient algorithms for constructing t-spanners and for solving the K-nearest-neighbour problem in higher dimensions are presented in [32]. In particular, it is shown that a K-th order θ-graph for a given point set in $\mathbb{R}^d$ can be computed in $\mathcal{O}\left(\text{sort}(N) + \frac{KN}{B}\log_{M/B}^{d-1}\frac{N}{B}\right)$ I/Os. A new construction based on an $\mathcal{O}(\sqrt{K})$-th order θ-graph is then used to compute the K closest pairs in $\mathcal{O}\left(\text{sort}(N) + \frac{N\sqrt{K}}{B}\log_{M/B}^{d-1}\frac{N}{B}\right)$ I/Os.

2 The Well-Separated Pair Decomposition

In this section, we recall the definition of a well-separated pair decomposition and related concepts.

For a given point set $S \subset \mathbb{R}^d$, the *bounding rectangle* $R(S)$ is the smallest rectangle containing all points in S, where a *rectangle* R is the Cartesian product $[x_1, x_1'] \times [x_2, x_2'] \times \cdots \times [x_d, x_d']$ of a set of closed intervals. The *length* of R in dimension i is $\ell_i(R) = x_i' - x_i$. The minimum and maximum lengths of R are $\ell_{\min}(R) = \min\{\ell_i(R) : 1 \leq i \leq d\}$ and $\ell_{\max}(R) = \max\{\ell_i(R) : 1 \leq i \leq d\}$, respectively. We call R a *box* if $\ell_{\max}(R) \leq 3\ell_{\min}(R)$. If all lengths of R are equal, R is a *cube*. We denote its side length by $\ell(R) = \ell_{\min}(R) = \ell_{\max}(R)$. Let $i_{\min}(R)$ be a dimension such that $\ell_{i_{\min}(R)}(R) = \ell_{\min}(R)$, and let $i_{\max}(R)$ be a dimension such that $\ell_{i_{\max}(R)}(R) = \ell_{\max}(R)$. For a point set S, let $\ell_i(S) = \ell_i(R(S))$, $\ell_{\min}(S) = \ell_{\min}(R(S))$, $\ell_{\max}(S) = \ell_{\max}(R(S))$, $i_{\min}(S) = i_{\min}(R(S))$, and $i_{\max}(S) = i_{\max}(R(S))$. We use $\text{dist}_2(p, q)$ to denote the Euclidean distance between two points p and q. This generalizes to point sets by defining $\text{dist}_2(P, Q) = \inf\{\text{dist}_2(p, q) : p \in P \text{ and } q \in Q\}$.

A *ball* of radius r and with center c is the point set $B = \{p \in \mathbb{R}^d : \text{dist}_2(c, p) \leq r\}$. Given a *separation constant* $s > 0$, we say that two point sets A and B are *well-separated* if $R(A)$ and $R(B)$ can be enclosed in two balls of radius r such that the distance between the two balls is at least sr (see Figure 1).

We define the *interaction product* of two point sets A and B as $A \otimes B = \{\{a, b\} : a \in$

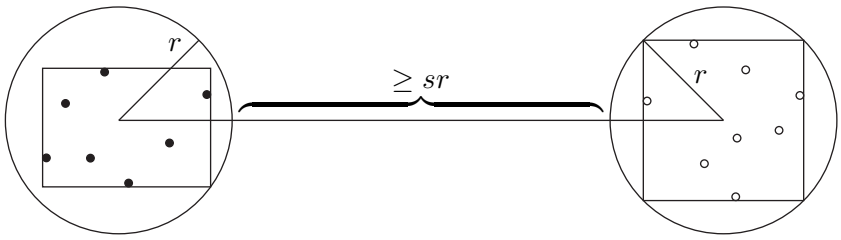

Figure 1. Two well-separated point sets A (black dots) and B (white dots).

$A \wedge b \in B \wedge a \neq b\}$. A *realization* $\mathcal{R}$ of $A \otimes B$ is a set $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ with the following properties:

(R1) $A_i \subseteq A$ and $B_i \subseteq B$, for $1 \leq i \leq k$,

(R2) $A_i \cap B_i = \emptyset$, for $1 \leq i \leq k$,

(R3) $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$, for $1 \leq i < j \leq k$, and

(R4) $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$.

Intuitively, this means that, for every pair $\{a, b\}$ of distinct points $a \in A$ and $b \in B$, there is a unique pair $\{A_i, B_i\}$ such that $a \in A_i$ and $b \in B_i$. A realization is *well-separated* if it has the following additional property:

(R5) Sets A_i and B_i are well-separated, for $1 \leq i \leq k$.

A binary tree T over the points in S defines a recursive partition of S into subsets in a natural manner: the leaves of T are in one-to-one correspondence to the points in S; an internal node v of T represents the set of points in S corresponding to the leaves of T that are descendants of v. We refer to a node that represents a subset $A \subseteq S$ as node A. A leaf that represents a point $a \in S$ is referred to as leaf a or node $\{a\}$, depending on the context. We define the *size* of a node $A \in T$ as the cardinality of set A. A realization of $A \otimes B$ *uses* a tree T if all sets A_i and B_i in the realization are nodes of T. A *well-separated pair decomposition* (WSPD) $\mathcal{D} = (T, \mathcal{R})$ of a point set S consists of a binary tree T over S and a well-separated realization $\mathcal{R}$ of $S \otimes S$ that uses T.

Two concepts that are useful when computing well-separated pair decompositions of point sets are those of fair splits and split trees. A *split* of a point set S is a partition of S into two non-empty point sets lying on either side of a hyperplane perpendicular to one of the coordinate axes and not intersecting any point in S. A *split tree* T of S is a binary tree over S defined as follows: If $S = \{x\}$, T consists of a single node $\{x\}$. Otherwise, consider a split that partitions S into two non-empty subsets S_1 and S_2; tree T then consists of a node S and two split trees T_1 and T_2 for S_1 and S_2, respectively, whose roots are children of S. For a node A in T, the *outer rectangle* $\hat{R}(A)$ is defined as follows: For the root S, let $\hat{R}(S)$ be a cube with side length $\ell(\hat{R}(S)) = \ell_{\max}(S)$ and centered at the center of $R(S)$.

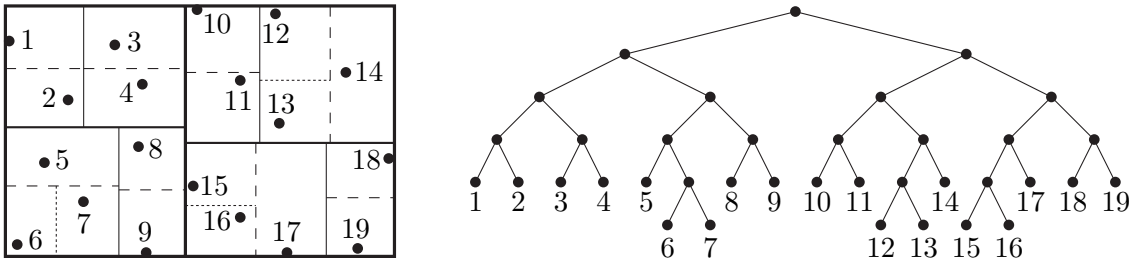

Figure 2. A partition of a given point set using fair splits and the corresponding fair split tree.

For all other nodes A, the hyperplane used for the split of A's parent, $p(A)$, divides $\hat{R}(p(A))$ into two rectangles. Let $\hat{R}(A)$ be the one that contains A. A *fair split* of A is a split of A where the hyperplane splitting A is at distance at least $\ell_{\max}(A)/3$ from each of the two sides of $\hat{R}(A)$ parallel to it. A split tree formed using only fair splits is called a *fair split tree* (see Figure 2). A *partial fair split tree* T is a subtree of a fair split tree T' that contains the root of T'; that is, the leaves of T may represent subsets of S instead of single points. Callahan and Kosaraju [15] use fair split trees to develop sequential and parallel algorithms for constructing a well-separated realization of $S \otimes S$ that consists of $\mathcal{O}(N)$ pairs of subsets of S.

The following are alternative, more restrictive, definitions of outer rectangles and fair splits that ensure in particular that all outer rectangles are boxes. Our algorithm for constructing a fair split tree makes sure that the splits satisfy these more restrictive conditions. For the root S of T, the outer rectangle $\hat{R}(S)$ is defined as above. Given the outer rectangle $\hat{R}(A)$ of a node, a split is fair if it splits $\hat{R}(A)$ perpendicular to its longest side and the splitting hyperplane has distance at least $\frac{1}{3}\ell_{\max}(\hat{R}(A))$ from the two sides of $\hat{R}(A)$ parallel to it.[1] Let R_1 and R_2 be the two rectangles produced by this split. The following procedure defines $\hat{R}(A_i)$, for $i \in \{1, 2\}$: If R_i can be split fairly, let $\hat{R}(A_i) = R_i$. Otherwise, split R_i perpendicular to its longest side so that the splitting hyperplane has distance at least $\frac{1}{3}\ell_{\max}(R_i)$ from each of the two sides of R_i parallel to it. Only one of the two resulting rectangles contains points in A_i. Repeat the process after replacing R_i with this non-empty rectangle. Effectively, $\hat{R}(A_i)$ is obtained by shrinking R_i.

3 Techniques

In this section, we discuss a number of techniques that will be used in our algorithms in subsequent sections. These techniques include the buffer tree [3], time-forward processing [17], and a buffered version of the topology tree [25].

[1] In [14], this type of split is called a *fair cut* to emphasize the difference to a regular fair split; every fair cut is a fair split.

3.1 The Buffer Tree

The *buffer tree* [3] is an extension of the well-known B-tree data structure [6]; it outperforms the B-tree in applications where a large number of updates (insertions and deletions) and queries need to be performed and immediate query responses are not required. Using the buffer tree, processing a sequence of N updates and queries takes $\mathcal{O}\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) = \mathcal{O}(\mathrm{sort}(N))$ I/Os. In [3], a priority queue based on the buffer tree is described. This priority queue can process any sequence of N Insert, Delete, and DeleteMin operations in $\mathcal{O}(\mathrm{sort}(N))$ I/Os.

3.2 Time-Forward Processing

Time-forward processing is a very elegant technique for solving graph problems that has been proposed in [17]. A more generally applicable and simpler implementation of this technique has later been provided in [3]. The following problem can be solved using this technique: Let G be a topologically sorted directed acyclic graph; that is, its vertices are arranged so that, for every edge (u, v) in G, vertex u precedes vertex v. Every vertex v of G has a priority, which is equal to its position in the sorted order, and a label $\phi(v)$. The goal is to compute a new label $\psi(v)$, for every vertex v, by applying a function f to $\phi(v)$ and the labels $\psi(u_1), \ldots, \psi(u_k)$ of the in-neighbours $u_1, \ldots, u_k$ of v: $\psi(v) = f(\phi(v), \psi(u_1), \ldots, \psi(u_k))$. To achieve this, the vertices of G are evaluated in topologically sorted order; after evaluating a vertex v, its label $\psi(v)$ is sent to all its out-neighbours, to ensure that they have v's label at their disposal when it is their turn to be evaluated. The implementation of this technique due to Arge [3] is simple and elegant: The "sending" of information is realized using a priority queue Q. When a vertex v wants to send its label $\psi(v)$ to another vertex w, it inserts $\psi(v)$ into priority queue Q and gives it priority w. When vertex w is being evaluated, it removes all entries with priority w from Q. Since every in-neighbour of w has sent its output to w by queueing it with priority w, this provides w with the required inputs. Moreover, every vertex removes its inputs from the priority queue before it is evaluated, and all vertices with smaller priorities are evaluated before w. Thus, the entries with priority w in Q are those with lowest priority at the time when w is evaluated and can therefore be removed using a sequence of DeleteMin operations. Using the buffer tree as priority queue, graph G can be evaluated in $\mathcal{O}(\mathrm{sort}(V + E + I))$ I/Os, where I is the total amount of information sent along the edges of G. (I may be $\omega(V + E)$ if labels $\phi(v)$ and $\psi(v)$ are allowed to be of non-constant size.)

3.3 The Topology Buffer Tree

The topology tree, introduced by Frederickson [25], is a data structure to represent dynamically changing, possibly unbalanced binary trees so that updates and queries of the tree can be performed in $\mathcal{O}(\log N)$ time. The topology tree $\mathcal{T}$ of a rooted binary tree T is defined as follows: A *cluster* C in T is the vertex set of a connected subgraph T' of T. The *root* of cluster C is the same as the root of tree T'. Two disjoint clusters C_1 and C_2 are adjacent

if there exist two vertices $v \in C_1$ and $w \in C_2$ that are adjacent in T. Cluster C_1 is a *child* of cluster C_2 if the parent of the root r_1 of C_1 is a node of C_2. A *restricted cluster partition* of T is a partition of the vertex set of T into disjoint clusters so that the following conditions hold:

(i) No cluster contains more than two vertices,

(ii) A cluster containing two vertices has at most one child, and

(iii) No two adjacent clusters can be combined without violating Condition (i) or (ii).

Given a binary tree T, a *multi-level cluster partition* of T is defined as a sequence $T = T_0, T_1, \ldots, T_r$ of binary trees such that, for all $1 \leq i \leq r$, tree T_i is obtained from tree T_{i-1} by contracting every cluster in a restricted cluster partition of T_{i-1} into a single vertex, and tree T_r has a single vertex. A *topology tree* $\mathcal{T}$ of tree T is obtained from a multi-level cluster partition of T as follows: The vertex set of $\mathcal{T}$ is the disjoint union of the vertex sets of trees $T_0, \ldots, T_r$. The vertices of tree T_i are at level i in $\mathcal{T}$, where level 0 is the level of the leaves of $\mathcal{T}$ and level r is the level of the root of $\mathcal{T}$. A vertex $v \in T_i$ is the parent of a vertex $w \in T_{i-1}$ in $\mathcal{T}$ if the cluster in the cluster partition of T_{i-1} represented by v contains vertex w. Since no cluster has size greater than two, $\mathcal{T}$ is a binary tree.

Lemma 1 (Frederickson [25]) *For all* $1 \leq i \leq r$, $|T_i| \leq \frac{5}{6}|T_{i-1}|$.

Lemma 1 implies that $r = \mathcal{O}(\log N)$ and $|\mathcal{T}| = \mathcal{O}(N)$. Next we characterize the kind of search queries that can be answered efficiently on a binary tree T using its topology tree $\mathcal{T}$. We call a query q *oblivious* w.r.t. T if the information stored at every node $v \in T$ is sufficient to conclude that either $T(v)$ or $T \setminus T(v)$ does not contain an answer to query q. Figure 3 illustrates that standard search queries on binary search trees are not oblivious. The next lemma shows that oblivious queries can be answered using a topology tree.

Lemma 2 *Given a topology tree $\mathcal{T}$ for a binary tree T, an oblivious query on T can be answered in $\mathcal{O}(\log N)$ time.*

Proof. For a node $v \in \mathcal{T}$, let $\mathrm{Desc}_0(v)$ be the set of vertices in T_0 that are descendants of v in $\mathcal{T}$, let T_v be the subgraph of T induced by the vertices in $\mathrm{Desc}_0(v)$, and let r_v be the root of T_v. We assume that every node $v \in \mathcal{T}$ stores the same information as r_v in T.

The query procedure is a simple extension of binary search: Consider the root u of $\mathcal{T}$, and let its two children be v and w. Then one of r_v and r_w, say r_v, is the root of T; trees T_v and T_w are obtained by removing the edge $(r_w, p(r_w))$ from T. The subtrees $\mathcal{T}(v)$ and $\mathcal{T}(w)$ of $\mathcal{T}$ rooted at v and w represent trees T_v and T_w, respectively. Since the query we ask is oblivious w.r.t. T, the information stored at r_w—that is, at w—is sufficient to decide whether T_w or $T \setminus T_w = T_v$ does not contain an answer to the query. In the former case, we look for an answer in T_v, by recursing on v. In the latter case, we look for an answer in T_w, by recursing on w.

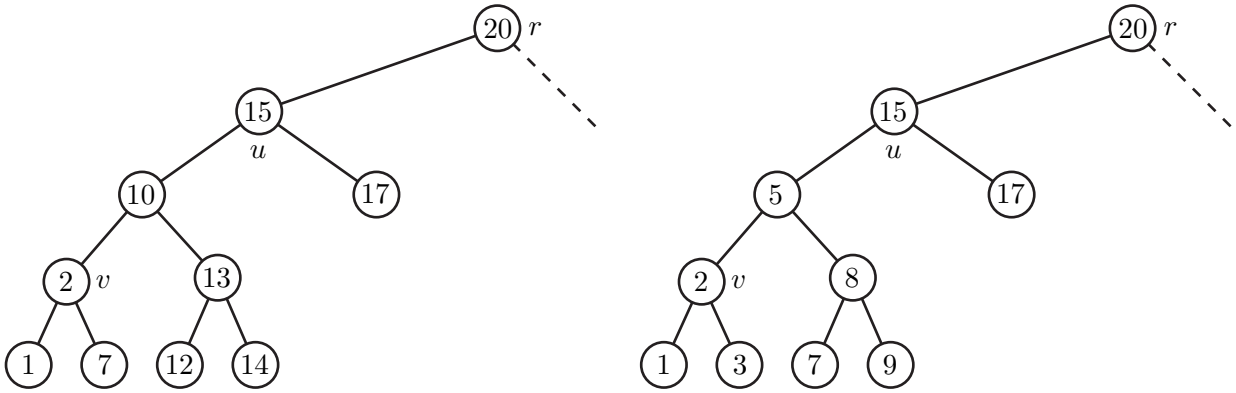

Figure 3. Two standard binary search trees. In the left tree, the answer to a search query for element 7 is stored in $T(v)$; in the right tree, the answer is stored in $T \setminus T(v)$. The information stored at node v is insufficient to distinguish between these two cases, because node v stores the same information in both trees. Hence, standard search queries are not oblivious w.r.t. standard binary search trees.

The correctness of this procedure is obvious. The running time is $\mathcal{O}(\log N)$ because we spend constant time per level of the topology tree, and there are $\mathcal{O}(\log N)$ levels, by Lemma 1. □

In [50], Zeh introduces a buffered version of the topology tree, which can be used to answer a batch of oblivious search queries I/O-efficiently. He builds a topology tree $\mathcal{T}$ for T and cuts it into layers of height $\log(M/B)$. The topology buffer tree $\mathcal{B}$ of T is obtained by contracting every subtree in such a layer into a single node. Tree $\mathcal{B}$ can be used to simulate searching $\mathcal{T}$: Filter all queries in the given set Q of queries through $\mathcal{B}$, from the root towards the leaves. For every node $v \in \mathcal{B}$, load the subtree $\mathcal{T}_v$ of $\mathcal{T}$ represented by v into internal memory and filter the queries in the set Q_v of queries assigned to v through $\mathcal{T}_v$ to determine for every query, the child of v to which this query is assigned next. By ensuring that the bottom-most trees in the partition of $\mathcal{T}$ have height $\log(M/B)$, it is guaranteed that $\mathcal{B}$ has size $\mathcal{O}(N/B)$ and height $\mathcal{O}(\log_{M/B}(N/B))$. The standard procedure for filtering queries through a buffer tree [3] then leads to the following result.

Theorem 1 (Zeh [50]) *A topology buffer tree $\mathcal{B}$ representing a binary tree T of size N can be constructed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. A batch of K oblivious search queries on T can then be answered in $\mathcal{O}(\mathrm{sort}(N+K))$ I/Os.*

In [50], a more complicated two-level data structure is used to achieve optimal performance on multiple disks. The problem in the multi-disk case is that the fan-out of the distribution procedure of [36] is only $\mathcal{O}(\sqrt{M/B})$. This would increase the number of nodes in $\mathcal{B}$ and thus the overhead I/Os involved in the distribution process. This problem is solved using the two-level scheme.

3.4 Querying a Hierarchy of Rectangles

We show how to use the topology buffer tree introduced in Section 3.3 to answer deepest containment queries on a hierarchy of nested rectangles represented by a binary tree T. Such queries need to be answered as part of our algorithm for constructing a fair split tree.

Let $S \subset \mathbb{R}^d$ be a point set, and let T be a binary tree with the following properties:

(i) Every node $v \in T$ has an associated rectangle $R(v)$,

(ii) For the root r of T, $R(r)$ contains all points in S,

(iii) For every node $v \neq r$ in T with parent $p(v)$, $R(v) \subseteq R(p(v))$, and

(iv) If $R(v) \cap R(w) \neq \emptyset$, then w.l.o.g. $R(v) \subseteq R(w)$ and w is an ancestor of v.

A *deepest containment query* on T is the following: *Given a point $p \in S$, find the node $v \in T$ such that $p \in R(v)$ and there is no descendant of v that satisfies this condition.* We show how to answer these queries for all points in S I/O-efficiently.

Lemma 3 *Given a set $S \subset \mathbb{R}^d$ of N points and a tree T with Properties (i)–(iv) above, deepest containment queries on T can be answered for all points $p \in S$ in $\mathcal{O}(\text{sort}(N + |T|))$ I/Os.*

Proof. Since the answer v to a query with a point $p \in S$ is stored in $T(w)$ if and only if $p \in R(w)$, deepest containment queries are oblivious w.r.t. tree T. Hence, we can apply Theorem 1 to answer deepest containment queries for all points in S in $\mathcal{O}(\text{sort}(N+|T|))$ I/Os. □

4 Constructing a Fair Split Tree

Our algorithm for computing a WSPD of a point set S is based on the algorithm of [15]. This algorithm uses the information provided by a fair split tree of S to derive the desired WSPD of S. Next we present an I/O-efficient algorithm to compute a fair split tree. We follow the framework of the parallel algorithm of [10]; but we do not simulate the PRAM algorithm, as this would lead to a higher I/O-complexity. First we outline the algorithm. Then we show that each of its steps can be carried out I/O-efficiently. Since we follow the framework of [10], the correctness of our algorithm follows from [10], provided that the presented I/O-efficient implementations of the different steps of the framework are correct.

To compute a fair split tree T of a point set S, we use Algorithm 1 and provide it with the set S and a cube R_0 that contains S. The side length of R_0 is $\ell(R_0) = \ell_{\max}(S)$. The centers of R_0 and $R(S)$ coincide. The algorithm computes the split tree T recursively. First it computes a partial fair split tree T' of S. Then it recursively computes fair split trees for the leaves of T'. In the rest of this section, we show that T' can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. As we show next, the leaves of T' are small enough to ensure that the total I/O-complexity of Algorithm 1 is $\mathcal{O}(\text{sort}(N))$.

Procedure FairSplitTree

Input: A point set $S \subset \mathbb{R}^d$ and a box R_0 that contains all points in S.
Output: A fair split tree T of S.

1: **if** $|S| \leq M$ **then**
2: Load point set S into internal memory and use the algorithm of [15] to compute T.
3: **else**
4: Apply procedure PartialFairSplitTree (Algorithm 2) to compute a partial fair split tree T' of S. The leaves of T' have size at most N^α.
5: Let $S_1, \ldots, S_k$ be the leaves of T'.
6: **for** $i = 1, \ldots, k$ **do**
7: Apply procedure FairSplitTree recursively to point set S_i and the outer rectangle $\hat{R}(S_i)$ of S_i in T', to compute a fair split tree T_i of S_i.
8: **end for**
9: $T = T' \cup T_1 \cup \cdots \cup T_k$.
10: **end if**

Algorithm 1. Computing a fair split tree of a point set.

Theorem 2 *Given a set S of N points in $\mathbb{R}^d$, Algorithm 1 computes a fair split tree T of S in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. By Lemma 4 below, procedure PartialFairSplitTree correctly computes a partial fair split tree T' of S. This implies that Algorithm 1 computes a fair split tree of S because it recursively augments T' with fair split trees of its leaves.

By Lemma 4, Line 4 of Algorithm 1 takes $\mathcal{O}(\text{sort}(N))$ I/Os. Thus, we obtain the following recurrence for the I/O-complexity of Algorithm 1:

$$\mathcal{I}(N) \leq c \cdot \text{sort}(N) + \sum_{i=1}^{k} \mathcal{I}(N_i),$$

where c is an appropriate constant such that Algorithm 2 takes at most $c \cdot \text{sort}(N)$ I/Os and N_i is the size of set S_i. Since $N_i \leq N^\alpha$ and $\sum_{i=1}^{k} N_i = N$, this can be expanded to

$$\mathcal{I}(N) \leq c \cdot \text{sort}(N) \sum_{i=0}^{\infty} \alpha^i,$$

whose solution is $\mathcal{I}(N) \leq \frac{c}{1-\alpha}\text{sort}(N) = \mathcal{O}(\text{sort}(N))$. Hence, Algorithm 1 takes $\mathcal{O}(\text{sort}(N))$ I/Os. □

In the rest of this section, we present the details of procedure PartialFairSplitTree (Algorithm 2) and prove the following lemma.

Procedure PartialFairSplitTree

Input: A point set $S \subset \mathbb{R}^d$ and a box R_0 that contains all points in S.
Output: A partial fair split tree T' of S whose leaves have size at most $|S|^\alpha$.

1: Compute a compressed pseudo split tree T_c of S such that none of the leaves of T_c has size more than $|S|^\alpha$.
2: Expand tree T_c to obtain a pseudo split tree T'' of S whose leaves have size at most $|S|^\alpha$.
3: Remove all nodes of T'' that do not represent any points in S. Compress every resulting path of degree-2 vertices into a single edge. The resulting tree is T'.

Algorithm 2. Computing a partial fair split tree of a point set S.

Lemma 4 *Given a set S of N points in $\mathbb{R}^d$, Algorithm 2 takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os to compute a partial fair split tree T' of S each of whose leaves represents a subset of at most N^α points in S, where $\alpha = 1 - \frac{1}{2d}$.*

Algorithm 2 computes the desired partial fair split tree T' in three phases. The first two phases produce a tree T'' that is almost a partial fair split tree, except that some of its leaves may represent boxes that are empty. We call T'' a *pseudo split tree*. The third phase removes these empty leaves and contracts T'' to obtain T'.

To construct T'', the first phase constructs a tree T_c that is a compressed version of T''. In particular, some leaves of T'' are missing in T_c and some edges of T_c have to be expanded to a tree that can be obtained through a sequence of fair splits. As we will see, attaching the missing leaves is easy; the compressed edges represent particularly nice sequences of fair splits that can be constructed I/O-efficiently. Next we provide the details of the three phases of Algorithm 2.

4.1 Constructing T_c

To construct tree T_c, we partition each dimension of rectangle R_0 into slabs such that each slab contains at most N^α points. We ensure that every leaf of T_c is contained in a single slab in at least one dimension. This guarantees that every leaf of T_c contains at most N^α points. The construction of these slabs is the only place where the construction of T_c depends on the point set S. Once the slabs are given, we consider the nodes of T_c to be rectangles produced from rectangle R_0 using fair splits.[2] The slabs are bounded by $\lceil N^{1-\alpha} \rceil + 1$ axes-parallel hyperplanes. We use these slab boundaries to guide the splits in the construction of T_c, thus limiting the number of rectangles that can appear as nodes of T_c. This is important for the following reason: Conceptually, we construct T_c by recursively splitting smaller and smaller rectangles, but we cannot afford to apply this sequential approach because it is not I/O-efficient. Instead, we generate all rectangles that might be nodes of T_c and construct a

[2]The splits are not fair in the exact sense of the definition because it is not guaranteed that there is at least one point on each side of the splitting hyperplane. All other conditions of a fair split are satisfied.

graph that has T_c as a subgraph. Then we extract T_c from this graph. The bounded number of nodes guarantees that the constructed graph has linear size and, thus, that T_c can be extracted efficiently.

Now consider a node $R \in T_c$ that needs to be split. Let R' denote the largest rectangle that is contained in R and whose sides lie on slab boundaries. To bound the number of rectangles we have to consider as candidates to be nodes of T_c, we maintain the following invariants for all rectangles R generated by the algorithm:

(i) In each dimension, at least one side of R lies on a slab boundary,

(ii) For all $i \in [1, d]$, either $\ell_i(R') = \ell_i(R)$ or $\ell_i(R') \leq \frac{2}{3}\ell_i(R)$, and

(iii) $\ell_{\min}(R) \geq \frac{1}{3}\ell_{\max}(R)$, that is, R is a box.

Invariants (i) and (ii) hold for rectangle R_0 by definition. Invariant (iii) also holds for R_0 because R_0 is either a cube or a rectangle computed by a previous invocation of the algorithm. Given a rectangle R that satisfies Invariants (i)–(iii), we apply one of the following cases (see Figure 4) to split it into two rectangles using a hyperplane perpendicular to dimension $i_{\max}(R)$. Each of these cases is said to "produce" one or two rectangles, which are the children of R in T_c and are subject to recursive splitting if necessary. The three cases ensure that the rectangles they produce satisfy Invariants (i)–(iii).

Case 1. $\ell_{\max}(R) = \ell_{i_{\max}(R)}(R')$, that is, both sides of R in dimension $i_{\max}(R)$ lie on slab boundaries. We find the slab boundary in this dimension that comes closest to splitting R into equal halves. If this boundary is at distance at least $\frac{1}{3}\ell_{\max}(R)$ from both sides of R in dimension $i_{\max}(R)$ (Case 1a), we split rectangle R along this slab boundary. Otherwise (Case 1b), we split rectangle R into two equal halves in dimension $i_{\max}(R)$. This case produces the two rectangles on both sides of the split.

If rectangle R does not satisfy Case 1, then $\ell_{i_{\max}(R)}(R') \leq \frac{2}{3}\ell_{\max}(R)$, that is, only one side of R in dimension $i_{\max}(R)$ lies on a slab boundary. We call this slab boundary H.

Case 2. $\ell_{\max}(R') \geq \frac{4}{27}\ell_{\max}(R)$. If $\ell_{i_{\max}(R)}(R') \geq \frac{1}{3}\ell_{\max}(R)$ (Case 2a), we split rectangle R along the hyperplane that contains the side of R' opposite to the one contained in H. Otherwise (Case 2b), we split rectangle R along a hyperplane at distance $y = \frac{2}{3}\left(\frac{4}{3}\right)^j \ell_{\max}(R')$ from H, where j is the unique integer such that $\frac{1}{2}\ell_{\max}(R) < \frac{2}{3}\left(\frac{4}{3}\right)^j \ell_{\max}(R') \leq \frac{2}{3}\ell_{\max}(R)$. Note that $0 \leq j \leq -\lfloor \log \frac{4}{27} / \log \frac{4}{3} \rfloor$, that is, there are only $\mathcal{O}(1)$ choices for j. This case produces only the rectangle containing R'. The reason for not including this second rectangle as a node of T_c is that it violates Invariant (i) in Case 2b. However, the second rectangle contains at most N^α points, as it is contained in a single slab of rectangle R_0. Hence, we can attach it as a leaf of T'' in the second phase without violating the constraint that no leaf of T'' contains more than N^α points.

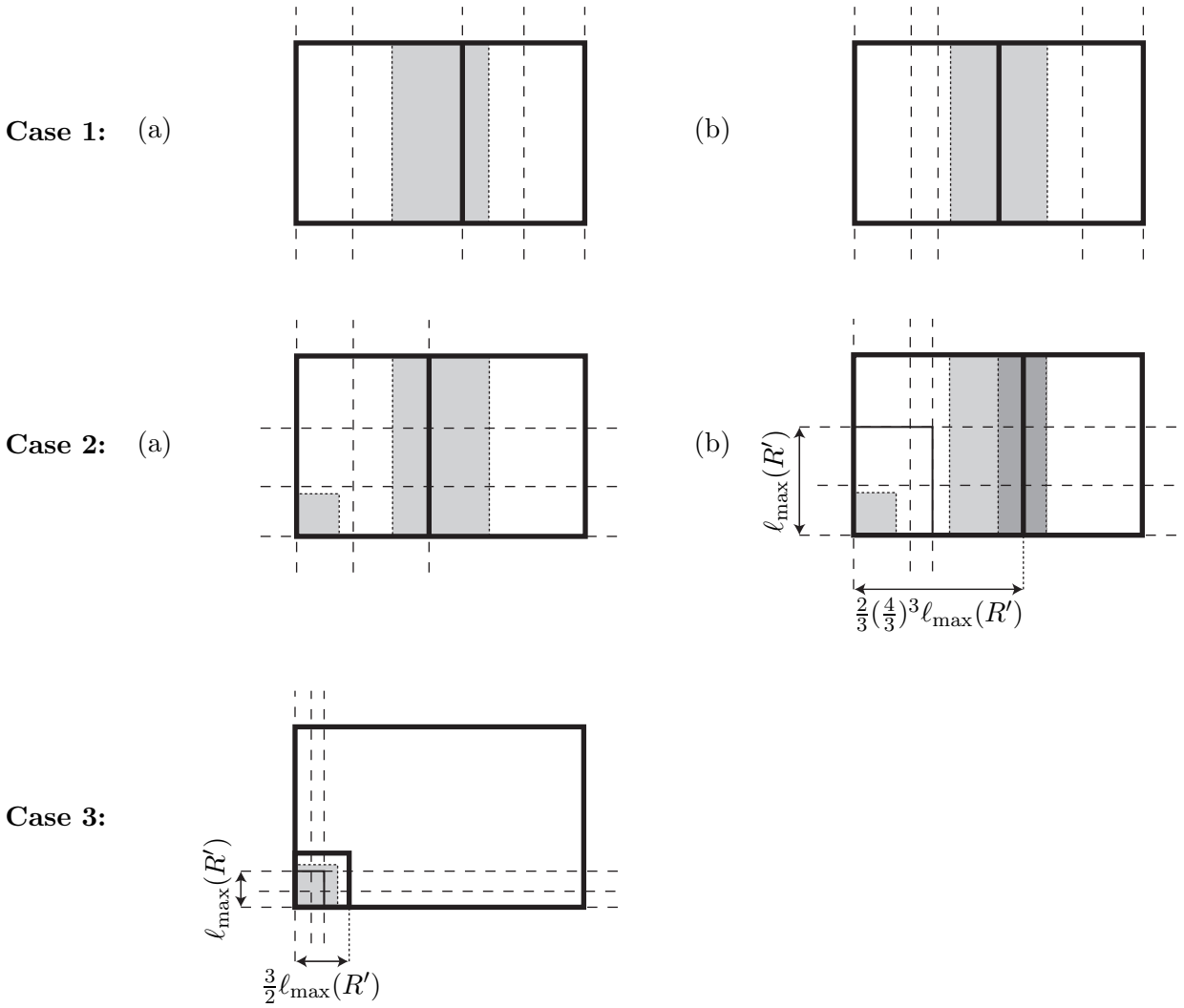

Figure 4. The different cases of the rectangle splitting rule. The thick rectangle is R. In Cases 1 and 2, the thick vertical line is the line where R is split. Slab boundaries are shown as dashed lines. Where shown, R' is shown as a thin solid rectangle. In Cases 1 and 2, the middle third of R is shaded. If there is a slab boundary passing through this region, we have Case 1a or Case 2a, respectively. In Case 2b, the splitting line is chosen so that it is in the darker right half of the shaded region. The shaded square in the bottom left corner in Cases 2 and 3 has side length $\frac{4}{27}\ell_{\max}(R)$. If there is a slab boundary that passes through R, but not through this region, we have Case 2. Otherwise, we have Case 3.

Case 3. $\ell_{\max}(R') < \frac{4}{27}\ell_{\max}(R)$. Then R' shares a unique corner with R. We construct a cube C that contains R', shares the same corner with R and R', and has side length $\ell(C) = \frac{3}{2}\ell_{\max}(R')$. This case produces the rectangle C. The edge between R and C is a *compressed edge*, which has to be expanded during the construction of T'' to obtain a sequence of fair splits that produce C from R. Again, the reason for introducing this compressed edge is that the rectangles produced by this sequence of fair splits would violate Invariant (i).

It is shown in [10] that each of the rectangles produced by these three cases satisfies Invariants (i)–(iii). Callahan also observes that each rectangle side produced by a split can be uniquely described using a constant number of slab boundaries in this dimension plus a constant amount of extra information. Since this is also true for the initial rectangle R_0 (because it has all its sides on slab boundaries), any rectangle that can be obtained from R_0 through a sequence of applications of the above rules can be described using two slab boundaries and a constant amount of information per dimension. The converse may not be true; that is, there may be rectangles that can be described in this way, but cannot be obtained from R_0 through a sequence of splits as defined in the above rules. The idea of Callahan's parallel algorithm for constructing T_c, which is also the basis for our I/O-efficient construction, is to construct the set of all rectangles that can be described in this fashion—Callahan calls these *constructible*—and then to extract the set of rectangles that *can* be produced from R_0. The key to the efficiency of the construction is the following lemma, which gives a bound on the total number of constructible rectangles. We note that a slightly looser bound is proved in [10].

Lemma 5 *There are* $\mathcal{O}\left(N^{2d(1-\alpha)}\right)$ *constructible rectangles.*

Proof sketch. Following the argument in [10], we prove that every dimension of a constructible rectangle can be represented using at most two slab boundary and an integer that is upper-bounded by a constant. Since there are only $\mathcal{O}(N^{1-\alpha})$ slab boundaries in each dimension, the lemma follows.

As in [10], we prove our claim for Case 2b because this is the case that can produce the largest number of rectangles. So consider splitting a rectangle R by applying Case 2b to its maximal dimension $i_{\max}$. Then the rectangle produced by this case is uniquely described by the slab boundary H that contains one of the two sides of R perpendicular to dimension $i_{\max}$, the length $\ell_{i_{\max}}(R')$ of R' in this dimension, and the integer j. The length $\ell_{i_{\max}}(R')$ is uniquely described by the two slab boundaries that contain the sides of R' perpendicular to dimension $i_{\max}$. Callahan concludes from this that 3 slab boundaries and the value of j are sufficient to describe the sides of the produced rectangle perpendicular to $i_{\max}$. To improve the bound to 2 slab boundaries, we observe that one of the slab boundaries defining $\ell_{i_{\max}}(R')$ is H. Thus, we can describe the produced rectangle using 2 slab boundaries, the integer j, and a bit that indicates which of the two slab boundaries is H. □

We apply the above splitting rules to obtain tree T_c as follows: The root of T_c is rectangle R_0. This node has two children, as rectangle R_0 always satisfies Case 1. For each of the children, we recursively construct a compressed pseudo split tree until all rectangles

have been split into rectangles that are not intersected by any slab boundary in at least one dimension. These rectangles are the leaves of T_c. Since a rectangle completely contained in a slab contains at most N^α points, it is guaranteed that no leaf in T_c has size more than N^α.

As mentioned above, constructing tree T_c using repeated fair splits does not lead to an I/O-efficient algorithm for constructing T_c. Hence, we extract tree T_c from a graph G whose nodes are all the constructible rectangles and which contains T_c as a subgraph. Graph G is defined as follows: There is a directed edge (R_1, R_2) in G, where R_1 and R_2 are two constructible rectangles, if rectangle R_2 is produced from rectangle R_1 by applying the above rectangle splitting rules. This implies that T_c is the subgraph of G containing all nodes R that can be reached from R_0 in G.

It is not hard to see that graph G is a DAG and that R_0 is one of its sources. Every vertex in G has out-degree at most two, which guarantees that the total size of graph G is linear in the number of constructible rectangles. If we choose $\alpha = 1 - \frac{1}{2d}$, Lemma 5 ensures that G has size $\mathcal{O}(N)$. In order to be able to extract T_c from G using the time-forward processing technique, we need an I/O-efficient procedure to topologically sort G. This is less trivial than it seems because no generally I/O-efficient algorithm for topologically sorting sparse DAGs is known. The solution we propose is based on the following observation.

Observation 1 *Let R_1 and R_2 be the two rectangles produced by splitting a rectangle R. Then $\sum_{j=1}^{d} \ell_j(R_i) < \sum_{j=1}^{d} \ell_j(R)$, for $i \in \{1, 2\}$.*

Since graph G contains an edge (R_1, R_2) only if rectangle R_2 is one of the rectangles produced by a split of R_1 or it is contained in one of the rectangles produced by a split of R_1, Observation 1 implies that G can be topologically sorted by sorting its vertices by the sums of their side lengths, in decreasing order. We are now ready to present the algorithm for computing T_c. The algorithm consists of three steps: Step 1 computes the slabs into which rectangle R_0 is partitioned. Step 2 constructs graph G from these slab boundaries. Step 3 extracts tree T_c.

Step 1—Constructing the slabs. To compute the slabs, we iterate over all d dimensions. For each dimension, we sort the points in S by their coordinates in this dimension. We scan the sorted list of points and place a slab boundary between the (jN^α)-th and $(jN^\alpha + 1)$-st point, for $1 \leq j \leq \lfloor N^{1-\alpha} \rfloor$. In addition, we place slab boundaries that coincide with the sides of R_0 perpendicular to the current dimension. Since we scan and sort set S once per dimension, the construction of the slabs in all dimensions takes $\mathcal{O}(d \cdot \text{sort}(N)) = \mathcal{O}(\text{sort}(N))$ I/Os.

Step 2—Constructing graph G. As mentioned before, every constructible rectangle can be encoded using two slab boundaries plus a constant amount of information per dimension. For instance, the i-th dimension of a rectangle produced by a type-1b split can be fully represented by the two slab boundaries of the original rectangle plus two flags saying that the rectangle is the result of a type-1b split and which of the two produced rectangles this is. Given that rectangles can be represented in this manner, we construct the vertex set of

graph G using $\mathcal{O}(d)$ nested scans. In particular, every vertex (that is, constructible rectangle) is a $3d$-tuple, storing 2 slab boundaries and a constant amount of information per dimension. Assume that every tuple is of the form $(s_1, s'_1, \alpha_1, s_2, s'_2, \alpha_2, \ldots, s_d, s'_d, \alpha_d)$, where s_i, s'_i, and α_i are the two slab boundaries and the extra information defining the corresponding rectangle in dimension i. To generate these tuples in lexicographically sorted order, we first create a list containing the right number of tuples with their entries uninitialized. A first scan over this list of tuples and the list of slab boundaries in the first dimension allows us to fill in the s_1-values of all tuples. In a second scan, we fill in the s'_1-values. More precisely, the second time around, we scan the list of tuples once, while we scan the list of slab boundaries once per slab boundary in the s_1-dimension of the tuples. The α_1-values can be filled in in another scan of the list of tuples without requiring a scan of slab lists. To fill in the s_2-values, we scan the list of tuples and the list of slab boundaries in the second dimension, where we scan the list of tuples once and the list of slab boundaries once per unique tuple (s_1, s'_1, α_1) in the list of partially defined tuples. We continue this until we have filled in all $3d$ dimensions of all tuples. Note that filling in one dimension of the tuples costs one scan of the tuple list and multiple scans of the corresponding slab list. However, the total number of items read from the slab list is equal to the number of tuples, so that the cost of filling in one dimension is equivalent to scanning the list of tuples twice. Since there are $\mathcal{O}(N)$ tuples and d is assumed to be constant, the total cost of constructing all rectangles is $\mathcal{O}(\text{scan}(N))$.

Every node of G now stores a complete representation of the rectangle R it represents. Before constructing the edge set of G, we augment every node with a complete description of the largest rectangle R' that is contained in R and is bounded by slab boundaries. To do this, we iterate over all dimensions and compute the boundaries of all rectangles R' in this dimension. To do the latter, we build a buffer tree over the slab boundaries in this dimension. Finding the boundary of the rectangle R' contained in rectangle R now amounts to answering standard search queries on the constructed tree. Since there are $\mathcal{O}\left(N^{1-\alpha}\right)$ slab boundaries and $\mathcal{O}(N)$ rectangles R, these queries can be answered in $\mathcal{O}(\text{sort}(N))$ I/Os per dimension. Since we assume that d is constant, the construction of rectangles R' takes $\mathcal{O}(d \cdot \text{sort}(N)) = \mathcal{O}(\text{sort}(N))$ I/Os in total.

To construct the edge set of G, we need to find at most two out-edges for each rectangle R. Since the dimensions of rectangles R and R' are stored locally with R, we can distinguish between Cases 1, 2, and 3 based only on the information stored with node R. The information is also sufficient to distinguish between Cases 2a and 2b, so that in Cases 2 and 3, we can construct the out-edge of rectangle R only from the information stored with R. To distinguish between Cases 1a and 1b, we need to find the slab boundary that comes closest to splitting R in half in dimension $i_{\max}(R)$. To do this, we apply the same approach as for computing rectangles R': We build a buffer tree over the slab boundaries in each dimension. Then we query this buffer tree with the position of the hyperplane H' that splits rectangle R in half. This query produces the positions of the two slab boundaries on each side of the hyperplane; it is easy to select the one that is closer to H'. Again, this involves answering $\mathcal{O}(N)$ queries on buffer trees of size $\mathcal{O}\left(N^{1-\alpha}\right)$. Hence, the edge set of G can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os.

Instead of adding edges as separate records, it is more convenient to represent the edges implicitly by storing all vertices as triples (R, R_1, R_2), where R_1 and R_2 are the two rectangles produced by the split of R. If the split of R produces only one rectangle R_1, we represent R as the triple $(R, R_1, \textbf{null})$. If R is a leaf, we represent R as the triple $(R, \textbf{null}, \textbf{null})$.

Step 3—Extracting T_c. Given graph G, as constructed in Step 2 of the algorithm, the extraction of T_c is rather straightforward. In particular, graph G can be topologically sorted in $\mathcal{O}(\text{sort}(N))$ I/Os, using Observation 1. Then we label every node except R_0 as inactive. Node R_0 is labelled as active. We apply time-forward processing to relabel the nodes of G. In particular, every active node remains active and sends "activate" messages to its out-neighbours. An inactive node that receives an "activate" message along one of its in-edges becomes active and forwards the "activate" message along its out-edges. Since graph G has size $\mathcal{O}(N)$, this application of time-forward processing takes $\mathcal{O}(\text{sort}(N))$ I/Os. Once the algorithm finishes, every node reachable from R_0 is active, while all other nodes are inactive. However, we have observed above that a node is in T_c if and only if it is reachable from R_0 in G. Hence, a final scan of the vertex set of G suffices to extract the vertices of tree T_c. This takes another $\mathcal{O}(\text{scan}(N))$ I/Os.

We have shown that each of the three steps of the construction of T_c can be carried out in $\mathcal{O}(\text{sort}(N))$ I/Os. Hence, we obtain the following lemma.

Lemma 6 *Given a point set $S \subset \mathbb{R}^d$ and a rectangle R_0 that encloses S, a compressed pseudo split tree of S with root R_0 can be constructed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

4.2 Constructing T''

The goal of the second phase of Algorithm 2 is to construct a pseudo split tree T'' of S from the compressed pseudo split tree T_c constructed in Phase 1. Tree T'' may have some leaves that do not correspond to any point in S; but, apart from that, it is a partial fair split tree whose leaves represent point sets of size at most N^α. In order to obtain T'' from T_c, we need to do the following:

(1) Attach the leaves that were discarded in Case 2 because they violate Invariant (i). It is easy to see that every such leaf is completely contained in a slab. Hence, it contains at most N^α points and just attaching it is sufficient; no splits are required.

(2) Replace every compressed edge (R, C) of T_c produced by an application of Case 3 with a sequence of fair splits that produce the shrunk cube C from rectangle R. Note that such a sequence of fair splits exists because $\ell(C) = \frac{3}{2}\ell_{\max}(R') < \frac{2}{9}\ell_{\max}(R) \leq \frac{2}{3}\ell_{\min}(R)$ (see the discussion of Step 2 below).

(3) Distribute the points of S to the leaves of T'' that contain them. This is necessary to recognize and discard empty leaves in Phase 3 of Algorithm 2. The recursive construction of fair split trees for the leaves of the partial fair split tree we compute also requires the point set contained in each leaf.

Next we show how to carry out these tasks in an I/O-efficient manner.

Step 1—Attaching missing leaves. Recall that every node R in T_c stores a full description of rectangles R and R'. As we argued in Section 4.1, this is sufficient to distinguish between Cases 1, 2, and 3 and to compute the rectangle produced by a type-2 split. Given a type-2 rectangle R and the rectangle R_1 produced by the type-2 split, the discarded rectangle is $R_2 = R \setminus R_1$. Since the information stored with every such rectangle R is sufficient to compute R_2, a single scan of the vertex set of T_c suffices to produce all discarded nodes. In particular, for every type-2 node R represented by the triple $(R, R_1, \textbf{null})$, we change its triple to (R, R_1, R_2) and add node R_2 to the vertex set of T_c. This takes $\mathcal{O}(\text{scan}(|T_c|)) = \mathcal{O}(\text{scan}(N))$ I/Os. We call the resulting tree T_c^+.

Step 2—Distributing the points of S and expanding compressed edges. As a result of the previous step, every point in S is contained either in some box that is a leaf of tree T_c or in a region $R \setminus C$, where (R, C) is a compressed edge produced by Case 3. Before expanding all compressed edges, we need to determine the region that contains each of the points in S. This is equivalent to answering deepest containment queries on T_c^+, for all points in S. By Lemma 3, this takes $\mathcal{O}(\text{sort}(N))$ I/Os.

To replace every compressed edge (R, C) produced by Case 3, we simulate one phase of the internal memory algorithm of [15] for constructing a fair split tree. This produces a tree $T(R, C)$ whose leaves form a partition of R into boxes. One of these leaves is C. All internal nodes of $T(R, C)$ are ancestors of C; that is, tree $T(R, C)$ is a path from R to C with an extra leaf attached to every node on this path except C. Note that every leaf $R' \neq C$ of $T(R, C)$ contains at most N^α points because it is completely contained between two slab boundaries in dimension $i_{\max}(p(R'))$. Hence, replacing every edge (R, C) in T_c^+ with its corresponding tree $T(R, C)$ produces a pseudo split tree T'' of S. It remains to show how to construct tree $T(R, C)$ I/O-efficiently.

We iterate the following procedure, starting with $R' = R$, until $R' = C$: If $\ell_{\max}(R') > 3\ell(C)$, we apply a split in dimension $i_{\max}(R')$ to produce two rectangles R_1 and R_2 of equal size. Otherwise, observe that R' and C share one side that is perpendicular to dimension $i_{\max}(R')$. Let H be the hyperplane that contains the other side of C perpendicular to dimension $i_{\max}(R')$, and let R_1 and R_2 be the two rectangles produced by splitting R' along hyperplane H. In both cases, let R_1 be the rectangle that contains C. It is not hard to verify that both R_1 and R_2 are boxes and that either $\ell_i(R_1) = \ell(C)$ or $\ell_i(R_1) \geq \frac{3}{2}\ell(C)$, for every dimension $1 \leq i \leq d$. The latter condition guarantees that the procedure can be applied iteratively to rectangle R_1. If rectangle R_2 contains at least one point, we distribute the points in $R' \setminus C$ to rectangles R_1 and R_2, make rectangles R_1 and R_2 children of rectangle R', and repeat the process with $R' = R_1$. Otherwise, we simply replace rectangle R' with R_1, essentially shrinking rectangle R', and repeat. This guarantees that only $\mathcal{O}(N)$ splits are performed for all compressed edges (R, C) in T_c^+. Also observe that all performed splits are fair: this is obvious if we split R' in half; if we do not split R' in half, we have $\frac{1}{3}\ell_{\max}(R') \leq \ell(C) \leq \frac{2}{3}\ell_{\max}(R')$, which makes a split along C's side fair.

To carry out this procedure, we use d sorted lists $L_1, \ldots, L_d$, where list L_i stores the points in $R \setminus C$ sorted by their i-th coordinates. When splitting rectangle R' in dimension $i_{\max} = i_{\max}(R')$, we scan list $L_{i_{\max}}$ from the appropriate end until we find the first point in R_1. If this is the first point in $L_{i_{\max}}$, we perform a shrink operation, since rectangle R_2 is empty. Otherwise, we remove the scanned points from list $L_{i_{\max}}$ and add them to the point set of leaf R_2. Before applying the algorithm to R_1, we have to remove the points in R_2 from all lists L_i, $i \neq i_{\max}$. Unfortunately, this may be expensive, because there is no guarantee that these points are stored consecutively in these lists. Hence, we do not modify lists L_i, $i \neq i_{\max}$, at this point. Instead, when using such a list L_i to perform a subsequent split, we augment the scan as follows: For every scanned point, we test whether it is contained in R_2. If this is the case, we append it to the point set of leaf R_2 as above. Otherwise, this point is contained in $R \setminus R'$ and, hence, has been written to the point set of some leaf of $T(R, C)$ that has been produced before. Thus, we remove the point from L_i without further action. In total, every list is scanned at most once, so that $\mathcal{O}(\text{scan}(|S \cap (R \setminus C)|))$ I/Os are sufficient to compute tree $T(R, C)$. In total, the replacement of all edges (R, C) with trees $T(R, C)$ takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Observe that the resulting tree T'' has size $\mathcal{O}(N)$: Tree T_c has size $\mathcal{O}(N)$. In the first step of the construction of T'' from T_c, we add at most one child to every node of T_c. The second step adds two nodes per split performed during the construction of a tree $T(R, C)$. As argued above, we perform only $\mathcal{O}(N)$ splits; the claim follows. We have shown the following lemma.

Lemma 7 *Given a compressed pseudo split tree T_c of a set S of N points in $\mathbb{R}^d$, a pseudo split tree T'' of S such that every leaf represents at most N^α points in S can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

While the procedure we have described is I/O-efficient, there is no bound on the number of shrink operations (which incur no I/Os) we perform; that is, the procedure, as described above, may spend too much computation time in internal memory. To rectify this, we briefly sketch how to avoid performing more than a linear number of shrink operations: When we are about to perform a shrink operation in dimension i, we scan lists $L_1, \ldots, L_d$, to identify the first point in R' in each dimension. Denote these points by $p_1, \ldots, p_d$. Let R^* be the smallest rectangle that shares the corner shared by R and C and contains points $p_1, \ldots, p_d$ as well as cube C. Let $\ell'_i = \ell_i(R^*)$ if $\ell_i(R^*) \leq \frac{2}{3}\ell_i(R')$. Otherwise, let $\ell'_i = \ell_i(R')$. Let $\ell'_{\max} = \max\{\ell'_i : 1 \leq i \leq d\}$. We shrink R' to the following rectangle R'': Rectangle R'' shares the same corner with R and C as R'. We define $\ell_j(R'') = \ell(C)$ if $\ell'_{\max} \leq 3\ell(C)$ and $\ell'_j = \ell(C)$. Otherwise, let $\ell_j(R'') = \max(\ell'_j, \frac{3}{2}\ell(C), \frac{1}{3}\ell'_{\max})$. It is easy to see that R'' is a box and that, in each dimension j, either $\ell_j(R'') = \ell(C)$ or $\ell_j(R'') \geq \frac{3}{2}\ell(C)$. Hence, C can be obtained from R'' through a sequence of fair splits. Moreover, if $R'' \neq C$, then the next split of R'' in dimension $i_{\max}(R'')$ is an actual split of R'', that is, it will leave at least one point on the side that does not contain C. Hence, we perform at most one shrink operation per split.

4.3 Constructing T'

Given a pseudo split tree T'' of S, we obtain a partial fair split tree T' from T'' by removing all nodes R from T'' such that $R \cap S = \emptyset$ and compressing all paths of degree-2 nodes in the resulting tree. Given the list of pairs (p, R), $R \in T''$ and $p \in S \cap R$, produced by the previous two phases, we sort the pairs in this list by their second components. We sort the vertex set of T''. Now a single scan of these two sorted lists suffices to mark every leaf R of T'' such that $R \cap S = \emptyset$ as "remove" and all remaining leaves as "keep". Using time-forward processing, we process T'' from the leaves towards the root to mark every internal node as "keep", "contract", or "remove", depending on whether none, one, or both of its children are marked as "remove". In addition to these marks, we label every node R with its closest descendant $K(R)$ that is labelled "keep". In particular, $K(R) = R$ if R is itself labelled "keep", and $K(R) = K(R')$ if R is labelled "contract", where R' is the child of R that is not labelled "remove". Finally, for every node labelled "keep", we replace both of its children R_1 and R_2 with their corresponding descendants $K(R_1)$ and $K(R_2)$. At the end of this labelling procedure, the tree induced by all nodes labelled "keep" is tree T'. We scan the vertex set of T'' one more time to remove all nodes marked as either "remove" or "contract". All the techniques used in this procedure take $\mathcal{O}(\text{sort}(N))$ I/Os. Hence, this third phase of Algorithm 2 takes $\mathcal{O}(\text{sort}(N))$ I/Os. We have shown the following lemma, which completes the proof of Lemma 4.

Lemma 8 *Given a pseudo split tree T'' of a set $S \subset \mathbb{R}^d$ of N points, a partial fair split tree T' of S such that every leaf in T' represents at most N^α points in S can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

5 Constructing a WSPD

In this section, we describe an I/O-efficient algorithm to extract a WSPD of a point set S from a fair split tree T of S. We assume that every leaf p of T is labelled with the coordinates of point p. Every internal node A is labelled with its bounding rectangle $R(A)$. Every pair $\{A_i, B_i\}$ in the computed WSPD is represented as the pair of nodes in T rather than using a full representation of both sets because, otherwise, the output may have size $\Omega(N^2)$.

Our algorithm simulates the internal memory algorithm of [15], which uses the fair split tree to drive the generation of pairs. We show that this computation can be translated into a traversal of a DAG G of size $\mathcal{O}(N)$. This traversal can be performed using the time-forward processing technique. The difficulty is that we are not able to generate G beforehand, because the presence of edges in G is decided only while the algorithm runs. We could generate a supergraph of G that contains all edges that are potentially in G; but there are $\Omega(N^2)$ such edges in the worst case, so that this does not lead to an efficient algorithm. Next we define the DAG G and show that it can be generated efficiently while traversing it.

In order to define G, we need to consider the internal memory algorithm of [15] for computing a well-separated realization of a point set S from its fair split tree T. We call this procedure ComputeWSR; its pseudo-code is shown in Algorithm 3. In this algorithm,

Procedure ComputeWSR

Input: A point set $S \subset \mathbb{R}^d$ and a fair split tree T of S.
Output: A well-separated realization $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ of $S \otimes S$ that consists of $k = \mathcal{O}(|S|)$ pairs and uses T.

1: For every internal node of T with children A and B, add a pair $\{A, B\}$ to a set R.
2: $\mathcal{R} \leftarrow \emptyset$
3: **for** every pair $\{A, B\} \in R$ **do**
4: Remove pair $\{A, B\}$ from R.
5: $\mathcal{R} \leftarrow \mathcal{R} \cup$ FindPairs$(\{A, B\}, T)$
6: **end for**

Procedure FindPairs

Input: A fair-split tree T and a pair $\{A, B\}$ of siblings in T.
Output: A well-separated realization $\mathcal{R}'$ of $A \otimes B$.

1: $R' \leftarrow \{\{A, B\}\}$
2: $\mathcal{R}' \leftarrow \emptyset$
3: **while** R' is not empty **do**
4: Remove a pair $\{A, B\}$ from R'.
5: {Assume w.l.o.g. that $B \prec A$.}
6: **if** A and B are well-separated **then**
7: Add the pair $\{A, B\}$ to $\mathcal{R}'$.
8: **else**
9: Let A_1 and A_2 be the two children of A in T.
10: Add pairs $\{A_1, B\}$ and $\{A_2, B\}$ to R'.
11: **end if**
12: **end while**

Algorithm 3. Computing a well-separated realization from a fair split tree.

---

we define $A \prec B$ if either $\ell_{\max}(A) < \ell_{\max}(B)$ or $\ell_{\max}(A) = \ell_{\max}(B)$ and $\nu(A) < \nu(B)$, for an arbitrary, but fixed, postorder numbering ν of T.

It is not hard to see that the set R constructed in Line 1 of procedure ComputeWSR is a realization of $S \otimes S$. However, in general, this realization is not well-separated. To obtain a well-separated realization of $S \otimes S$, the algorithm iterates over all pairs in R and tests whether they are well-separated. If a pair $\{A, B\}$ is well-separated, it is added to realization $\mathcal{R}$. Otherwise, it is replaced by a well-separated realization of $A \otimes B$. Hence, the algorithm maintains the invariant that $R \cup \mathcal{R}$ is a realization of $S \otimes S$ and all pairs in $\mathcal{R}$ are well-separated. Since set R is empty when the algorithm terminates, the final set $\mathcal{R}$ is a well-separated realization of $S \otimes S$. Using packing arguments, Callahan and Kosaraju [15] show that the computed realization has size $\mathcal{O}(|S|)$. To show that the algorithm takes $\mathcal{O}(|S|)$ time

to produce realization $\mathcal{R}$, they introduce the concept of a *computation tree*. Each such tree represents the pairs $\{A, B\}$ produced in one invocation of procedure FindPairs.

Formally, a computation tree is defined as follows: Let $\{A, B\}$, $B \prec A$, be a pair of nodes in T. If A and B are well-separated, the computation tree $T(\{A, B\})$ has a single node (A, B). Otherwise, let A_1 and A_2 be the two children of A in the fair split tree T. Then tree $T(\{A, B\})$ consists of a root node (A, B) and two computation trees $T(\{A_1, B\})$ and $T(\{A_2, B\})$ whose roots are children of (A, B). The size of a computation tree is proportional to the number of its leaves. Each of these leaves corresponds to a pair in the computed well-separated realization. Thus, the total size of all computation trees $T(\{A, B\})$, $\{A, B\} \in R$, is $\mathcal{O}(|S|)$.

We are now ready to describe the DAG G whose traversal we use to simulate the computation of procedure ComputeWSR. The vertex set of G is the same as that of the given fair split tree T. There is an edge from a node A_1 to a node A_2 in G if there are two nodes (A_1, B_1) and (A_2, B_2) in a computation tree $T(\{A, B\})$, $\{A, B\} \in R$, such that (A_1, B_1) is the parent of (A_2, B_2) in this tree. Next we show that G is a DAG and that the computation of procedure ComputeWSR can be simulated using a traversal of G.

Lemma 9 *Graph G is a DAG.*

Proof. We show that, for every edge (A_1, A_2) in G, $A_2 \prec A_1$. Moreover, it is easy to verify that "$\prec$" is a total order. This implies that G is acyclic. To see that $A_2 \prec A_1$, observe that edge (A_1, A_2) is in G only because there is an edge $((A_1, B_1), (A_2, B_2))$ in a computation tree. This, however, implies that either $A_2 = B_1$ or A_2 is a child of A_1 in T. In the former case, $A_2 \prec A_1$, by definition. In the latter case, $\ell_{\max}(A_2) \leq \ell_{\max}(A_1)$ and $\nu(A_2) < \nu(A_1)$, so that again $A_2 \prec A_1$. □

The computation of procedure ComputeWSR can be simulated by a traversal of G as follows: Initially, we store every pair $\{A, B\} \in R$, $B \prec A$, with node $A \in G$. Then we process the nodes of G in topologically sorted order. For every node $A \in G$, we process the pairs stored with node A one by one. For every pair $\{A, B\}$, $B \prec A$, we check whether A and B are well-separated. If so, we add the pair to the well-separated realization. Otherwise, let (A', B') and (A'', B'') be the children of node (A, B) in the computation tree containing node (A, B). We "send" pair $\{A', B'\}$ along edge (A, A') and pair $\{A'', B''\}$ along edge (A, A'') to add these pairs to the sets of pairs stored with nodes A' and A''. By the definition of G, edges (A, A') and (A, A'') exist in G because edges $((A, B), (A', B'))$ and $((A, B), (A'', B''))$ exist in the computation tree.

Next we argue that the edge set of graph G does not have to be known in advance, in order to perform the above traversal of G using time-forward processing. To prepare graph G, we compute a postorder numbering of the nodes of T. We sort the nodes of T by the relation "$\prec$" defined by this postorder numbering and assign a number $\eta(A)$ to every node $A \in T$, which represents the position of node A in the sorted sequence of nodes. We store with every node A of T the labels $\eta(A_1)$ and $\eta(A_2)$ of its two children A_1 and A_2 in T. Finally, we sort the nodes of T by decreasing numbers $\eta(A)$. This preprocessing can be carried out in $\mathcal{O}(\text{sort}(|S|))$ I/Os, since computing a postorder numbering and copying the

label $\eta(A)$ of each node to its parent can be realized using time-forward processing. Besides that, we sort the vertex set of T twice.

To initiate the processing of G, we scan the list of nodes. For every internal node of T with children A and B, $B \prec A$, we insert the pair (A, B) into a priority queue Q and give it priority $\eta(A)$. Now we process the nodes of G in their order of appearance. For every node A, we retrieve all pairs (A, B) from Q, one by one. For every pair (A, B), we test whether A and B are well-separated. If so, we add pair $\{A, B\}$ to the realization. Otherwise, let A_1 and A_2 be the two children of A. (Recall that numbers $\eta(A_1)$ and $\eta(A_2)$ are stored with node A.) If $\eta(A_1) > \eta(B)$, we insert a pair (A_1, B) into priority queue Q and give it priority $\eta(A_1)$. Otherwise, we insert a pair (B, A_1) into priority queue Q and give it priority $\eta(B)$. We do the same for the other child A_2. This is equivalent to sending pairs $\{A_1, B\}$ and $\{A_2, B\}$ along the corresponding edges of G. Then we proceed to the next pair.

The procedure just described is the same as standard time-forward processing with the exception that every node constructs its out-neighbourhood depending on the data it receives from its in-neighbours. The crucial fact is that the constructed out-neighbourhood of a node A contains only nodes B with $\eta(B) < \eta(A)$; hence, every node sending a pair to node B does so before node B is processed by the above procedure.

We have to bound the number of priority queue operations performed. Note that we queue and dequeue only pairs (A, B) that correspond to nodes in the computation trees and that every such pair is queued and dequeued at most once. Hence, the total number of priority queue operations is at most twice the total size of all computation trees, which is $\mathcal{O}(|S|)$. This implies that our simulation of procedure ComputeWSR using a traversal of graph G takes $\mathcal{O}(\text{sort}(|S|))$ I/Os. By Theorem 2, a fair split tree of a point set S can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. Thus, we obtain the following result.

Theorem 3 *Given a set S of N points in $\mathbb{R}^d$, a well-separated pair decomposition of S consisting of $\mathcal{O}(N)$ pairs can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

6 Applications of the WSPD

6.1 t-Spanners

Given a set S consisting of N points in $\mathbb{R}^d$ and a real number $t > 1$, a *t-spanner* for S is a graph G with vertex set S such that any two vertices u and v are connected by a path in G whose length is at most $t \cdot \text{dist}_2(u, v)$. In [15], it is shown that the following graph $G = (S, E)$ is a t-spanner of linear size for a point set $S \subset \mathbb{R}^d$: Given a WSPD $\mathcal{D} = (T, \mathcal{R})$ of S with separation factor $s = 4\frac{t+1}{t-1}$ and consisting of $\mathcal{O}(|S|)$ pairs, choose a representative $r(A) \in A$, for every node A in the fair split tree T. For every pair $\{A_i, B_i\}$ in the realization $\mathcal{R}$, add an edge $\{r(A_i), r(B_i)\}$ to E.

To choose representatives for all nodes $A \in T$, we use time-forward processing to process T from the leaves towards the root. Every leaf has itself as a representative. Every internal node chooses one of the representatives of its two children as a representative. Hence, the

representatives for all nodes $A \in T$ can be computed in $\mathcal{O}(\text{sort}(|S|))$ I/Os. In order to create the edge set of the spanner G, we create a list L that contains a pair $(A, r(A))$ for every node A of T. We sort the pairs in L by their first components and do the same for all pairs in $\mathcal{R}$. Now we scan lists L and $\mathcal{R}$ simultaneously to replace the first component A of every pair $\{A, B\}$ with the representative $r(A)$ of A. We sort the pairs in $\mathcal{R}$ by their second components and repeat the scan of lists L and $\mathcal{R}$ to replace the second component B of every pair $\{A, B\}$ with its representative $r(B)$. This involves sorting and scanning two lists of size $\mathcal{O}(|S|)$ a constant number of times and, hence, takes $\mathcal{O}(\text{sort}(|S|))$ I/Os.

In [5], it is shown that one can construct a spanner of spanner diameter at most $2 \log N$ by choosing the representatives $r(A)$ above more carefully; that is, for any two points p and q, the resulting graph contains a t-spanner path from p to q consisting of at most $2 \log N$ edges. In particular, given the representatives $r(A_1)$ and $r(A_2)$ for the children A_1 and A_2 of A, we choose $r(A)$ to be the representative $r(A_i)$ of the heavier subtree $T(A_i)$, where the weight of a tree is the number of leaves in the tree. This modified rule for choosing representatives can easily be incorporated in the time-forward processing step used to compute representatives. Hence, we obtain the following result.

Theorem 4 *Given a set S of N points in $\mathbb{R}^d$, a t-spanner of linear size and spanner diameter at most $2 \log N$ for S can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Finally, the following result shows that Theorem 4 is optimal.

Theorem 5 *It takes $\Omega(\text{sort}(N))$ I/Os to compute a t-spanner of linear size for a set S of N points in $\mathbb{R}^d$ and any constant $t > 1$.*

Proof. Given a set $S = \{p_1, \ldots, p_N\}$ of points in $\mathbb{R}^d$, a t-spanner G of S has to contain an edge $\{p_i, p_j\}$, for every pair of points $p_i = p_j$, $i \neq j$. Hence, a single scan of the edge set of G is sufficient to decide whether all points in S are distinct. If G has linear size, this takes $\mathcal{O}(\text{scan}(N))$ I/Os. Thus, if G can be constructed in $o(\text{sort}(N))$ I/Os, the element uniqueness problem can be solved in $o(\text{sort}(N))$ I/Os, which contradicts the $\Omega(\text{sort}(N))$ I/O lower bound shown for this problem in [4]. □

Note that the proof of Theorem 5 applies only if it is not known whether there are two identical points in S. If this is known, we observe that, for spanning ratios $t \leq 2$, a t-spanner must contain an edge between the closest pair. Since an $\Omega(\text{sort}(N))$ lower bound is shown for this problem in [4], this implies that Theorem 5 holds for sets of distinct points as long as $t \leq 2$.

6.2 *K* Nearest Neighbours

In this section, we show how to compute the K nearest neighbours for every point p of a point set $S \subset \mathbb{R}^d$, that is, the K points in $S \setminus \{p\}$ that are closest to p. Our construction follows the internal-memory algorithm of [15] for this problem.

Lemma 10 (Callahan/Kosaraju [15]) *Let $\{A, B\}$ be a pair in a well-separated realization of $S \otimes S$ with separation $s > 2$. If there is a point $b \in B$ that is among the K nearest neighbours of a point $a \in A$, then $|A| \leq K$.*

Given a point set B, let O_B be the center of its bounding rectangle $R(B)$. We divide the space around O_B into a constant number of cones with apex O_B such that, for any two points a and a' in the same cone, the angle $\angle aO_Ba'$ between segments $\overline{O_Ba}$ and $\overline{O_Ba'}$ is less than $\alpha = \frac{s}{s+1}$. The existence of such a set of cones has been shown by Yao [49], who calls it an *α-frame*.

Lemma 11 (Callahan/Kosaraju [15]) *Let X and B be point sets such that, for every point $x \in X$, the pair $\{\{x\}, B\}$ is well-separated; let C be a cone with apex O_B such that, for any two points a and a' in C, $\angle aO_Ba' < \frac{s}{s+1}$; and let $X \cap C = (x_1, \ldots, x_l)$ be the set of points in X that lie in C, sorted by their distances from O_B. For $i > K$, no point in B can be among the K nearest neighbours of x_i.*

Based on Lemma 10, the algorithm of [15] first extracts all pairs $\{A_i, B_i\}$ with $|A_i| \leq K$. For a node B in T, let $\{A'_1, B\}, \ldots, \{A'_q, B\}$ be the set of pairs in the WSPD that contain B and such that $|A'_i| \leq K$. Note that the sets $A'_1, \ldots, A'_q$ are pairwise disjoint. The algorithm constructs a candidate set $X(B) = \bigcup_{i=1}^{q} A'_i$, for every node $B \in T$. Now the nodes of T are processed from the root toward the leaves. For every node $B \in T$, the algorithm creates an α-frame $\mathcal{C}$ around O_B, partitions the points in $X(B)$ into *cone sets* $X_C = X(B) \cap C$, $C \in \mathcal{C}$, and extracts the set X'_C of the K points in X_C that are closest to O_B. Let $N(B) = \bigcup_{C \in \mathcal{C}} X'_C$. From the two lemmas above, it follows that $N(B)$ contains all points $p \in S$ that have one of their K nearest neighbours in B. Now the candidate sets $X(B_1)$ and $X(B_2)$ of the two children B_1 and B_2 of B in T are augmented with the points in $N(B)$. Then the algorithm proceeds to the next node. Eventually, the procedure produces sets $N(b)$, $b \in S$, such that the points having b as one of their K nearest neighbours are contained in $N(b)$. The algorithm produces sets $N'(a)$, $a \in S$, such that $b \in N'(a)$ if and only if $a \in N(b)$. Finally, the K nearest neighbours of every point a are extracted from $N'(a)$ using K-selection.

The crucial observation used in [15] to show that this algorithm takes $\mathcal{O}(KN)$ time is the following: Initially, the candidate sets $X(B)$, $B \in T$, have total size $\mathcal{O}(KN)$ because there are only $\mathcal{O}(N)$ pairs in the given WSPD and every pair contributes at most K points to some set $X(B)$. When processing T top-down, each of the $\mathcal{O}(N)$ sets $X(B)$ is augmented with the $\mathcal{O}(K)$ points in $N(p(B))$, which adds another $\mathcal{O}(KN)$ points to the total size of all candidate sets $X(B)$. Hence, during the construction of sets $N(b)$, K-selection is applied to candidate sets of total size $\mathcal{O}(KN)$, which takes $\mathcal{O}(KN)$ time. In internal memory, sets $N'(a)$ are readily constructed in $\mathcal{O}\left(\sum_{b \in S} |N(b)|\right) = \mathcal{O}(KN)$ time, and the final application of K-selection to these sets takes $\mathcal{O}(KN)$ time again.

Once the initial candidate sets $X(B)$, $B \in T$, have been computed, the rest of the algorithm can easily be carried out in $\mathcal{O}(\mathrm{sort}(KN))$ I/Os. In particular, we realize the augmentation of every set $X(B)$ with the points in $N(p(B))$ using time-forward processing, sending the contents of set $N(p(B))$ from $p(B)$ to B. When processing node B, we append the received points to $X(B)$, partition the resulting set into the cone sets X_C, and apply an

I/O-efficient K-selection algorithm to each of the sets X_C; this takes $\mathcal{O}(\mathrm{sort}(|X(B)|))$ I/Os.[3] Hence, the total number of I/Os spent on constructing sets $N(b)$, for all leaves b of T, is $\mathcal{O}(\mathrm{sort}(KN))$: $\mathcal{O}(\mathrm{sort}(KN))$ I/Os to send sets $N(p(B))$ along the edges of T using time-forward processing, and $\mathcal{O}(\mathrm{scan}(KN))$ I/Os for all applications of K-selection.

If we represent every set $N(b)$ as a collection of pairs $\{(a, b) : a \in N(b)\}$, where b is a leaf of T, sets $N'(a)$, $a \in S$, can be constructed by sorting the union of these sets lexicographically. Then we apply K-selection to each of these sets $N'(a)$ to extract the K nearest neighbours of point a. This takes another $\mathcal{O}(\mathrm{sort}(KN))$ I/Os. It remains to show how to construct the initial candidate sets $X(B)$, $B \in T$, I/O-efficiently.

First we compute a postorder numbering ν of the nodes of T and a labelling of every node $A \in T$ with the number $\lambda(A)$ of leaves that are descendants of A in T. Similar to the conversion of the pairs $\{A_i, B_i\}$ in $\mathcal{R}$ into spanner edges $\{r(A_i), r(B_i)\}$, described in the previous section, we can generate a list Y with pairs $(A, \nu(A))$, $A \in T$, and then sort and scan Y and $\mathcal{R}$ a constant number of times to inform every pair $\{A, B\}$ in $\mathcal{R}$ about the postorder numbers $\nu(A)$ and $\nu(B)$ of its endpoints. Then we create two copies of every pair $\{A, B\}$ in the given WSPD, representing one as the ordered pair (A, B) and the other as the ordered pair (B, A). We sort the nodes of T according to the above postorder numbering and the ordered pairs (A, B) and (B, A) by the postorder numbers of their first components. We scan the sorted list of nodes and create a list L containing all leaves of T, sorted by increasing postorder numbers. Now we scan the list of nodes of T, the list of pairs in the WSPD, and the list of leaves of T. The scans of the latter two lists are driven by the information found in the node list as follows: For every node $A \in T$, we continue the scan of list L until a leaf a with $\nu(a) > \nu(A)$ is found. If $\lambda(A) > K$, we skip over all pairs (A, B) in the list of pairs. If $\lambda(A) \leq K$, we repeat the following for every pair (A, B) in the list of pairs: We scan backward from the current position in L to read the last $\lambda = \lambda(A)$ leaves $v_1, \ldots, v_\lambda$ in L and append pairs $(\nu(B), v_1), \ldots, (\nu(B), v_\lambda)$ to a list $\mathcal{X}$. Once all nodes of T have been processed, we sort the pairs in list $\mathcal{X}$ by their first components, thereby transforming list $\mathcal{X}$ into a concatenation of lists $X(B)$, for all $B \in T$.

The computation of labels $\nu(A)$ and $\lambda(A)$, for all nodes $A \in T$, can be carried out in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, using time-forward processing. Informing all pairs in $\mathcal{R}$ about the postorder numbers of their endpoints requires sorting and scanning lists of size $\mathcal{O}(N)$ a constant number of times and, hence, takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. Then the algorithm sorts three lists of size $\mathcal{O}(N)$, which takes $\mathcal{O}(\mathrm{sort}(N))$ I/Os. The scans used to produce list $\mathcal{X}$ scan a total of $\mathcal{O}(KN)$ data items, so that this takes $\mathcal{O}(\mathrm{scan}(KN))$ I/Os. The final sorting of list $\mathcal{X}$ takes $\mathcal{O}(\mathrm{sort}(KN))$ I/Os because list $\mathcal{X}$ has size $\mathcal{O}(KN)$. Hence, we obtain the following result.

Theorem 6 *Given a set S of N points in $\mathbb{R}^d$, the K nearest neighbours of every point in S can be computed in $\mathcal{O}(\mathrm{sort}(KN))$ I/Os.*

[3] It is easy to verify that the standard K-selection algorithm of [20] takes $\mathcal{O}(\mathrm{scan}(N))$ I/Os when applied to a set of size N.

6.3 K Closest Pairs

In this section, we show how to enumerate the K smallest inter-point distances in a point set $S \subset \mathbb{R}^d$. In [13], the following approach has been proposed to solve this problem: Given a WSPD $\mathcal{D}$ of S, let $\langle \{A_1, B_1\}, \ldots, \{A_q, B_q\} \rangle$ be the list of pairs in $\mathcal{D}$, sorted by increasing distances $\text{dist}_2(R(A_i), R(B_i))$. Find the smallest index i such that $\sum_{j=1}^{i} |A_j||B_j| \geq K$ and retrieve all pairs $\{A, B\}$ such that $\text{dist}_2(R(A), R(B)) \leq (1 + 4/s)r$, where $r = \text{dist}_2(R(A_i), R(B_i))$. Then generate the set C of all pairs $\{a, b\}$ such that $a \in A$ and $b \in B$, for some pair $\{A, B\}$ with $\text{dist}_2(R(A), R(B)) \leq (1+4/s)r$. Now apply K-selection to find the set X of K pairs such that $\text{dist}_2(a, b) \leq \text{dist}_2(a', b')$, for any $\{a, b\} \in X$ and $\{a', b'\} \in C \setminus X$.

The correctness of this solution has been shown in [13]. It is also shown in [13] that the set C has size $\mathcal{O}(N + K)$. Thus, once the set C has been produced, the application of K-selection to C takes $\mathcal{O}(\text{scan}(N + K))$ I/Os. We show that the set C can be computed in $\mathcal{O}(\text{sort}(N + K))$ I/Os.

First we use time-forward processing to process T from the leaves toward the root, computing for every node $A \in T$, the size $|A|$ of point set A. Using the same technique as in the previous two sections, we inform every pair $\{A, B\}$ about the cardinalities and bounding rectangles of sets A and B. We sort these pairs by their distances $\text{dist}_2(R(A), R(B))$ and scan the list of pairs by increasing distances and sum the cardinalities $|A||B|$ of the scanned pairs $\{A, B\}$. We stop as soon as this sum is at least K. Let $\{A_i, B_i\}$ be the pair where the scan stopped. We continue the scan until we find the first pair $\{A, B\}$ with $\text{dist}_2(R(A), R(B)) > (1 + 4/s)\text{dist}_2(R(A_i), R(B_i))$. We discard this pair as well as all remaining pairs in the list.

Now we make two copies of each remaining pair $\{A, B\}$, representing one as the ordered pair (A, B) and the other as the ordered pair (B, A). We sort these ordered pairs by their first components, thereby producing, for every node $A \in T$, the list of pairs $(A, B_1), \ldots, (A, B_{\rho(A)})$ among the remaining pairs. We use a similar procedure as the one described in Section 6.2 to extract, for every node A with $\rho(A) > 0$, the list of points in set A. For every pair (A, B_i), we make a copy of point set A, representing every point $a \in A$ as the triple $(\{A, B_i\}, A, a)$. We sort the resulting list of triples lexicographically. As a result, for every pair $\{A, B\}$, point sets A and B are stored consecutively. Two nested scans of these two sets suffice to create the set $\{\{a, b\} : a \in A \text{ and } b \in B\}$.

The extraction of all relevant pairs takes $\mathcal{O}(\text{sort}(N))$ I/Os, since we apply time-forward processing and sort and scan lists of size $\mathcal{O}(N)$ a constant number of times. Given the extracted pairs, producing lists $(A, B_1), \ldots, (A, B_{\rho(A)})$, for all nodes $A \in T$, takes $\mathcal{O}(\text{sort}(N))$ I/Os, since there are only $\mathcal{O}(N)$ pairs to be sorted. Using the same arguments as in Section 6.2, the I/O-complexity of the extraction of pairs of sets $\{A, B\}$ can be bounded by $\mathcal{O}(\text{sort}(N + K))$ because the cardinality of all extracted sets is $\mathcal{O}(N + K)$. Finally, the scan to produce the candidate set of point pairs takes $\mathcal{O}(\text{scan}(N + K))$ I/Os because $\mathcal{O}(N + K)$ is a bound on the cardinality of the produced set. Hence, we obtain the following result.

Theorem 7 *Given a set S of N points in $\mathbb{R}^d$, the K closest pairs in S can be found in $\mathcal{O}(\text{sort}(N + K))$ I/Os.*

7 A Note on Dynamic WSPD and Closest Pair

We want to point out that, using the topology B-tree of [12] and ideas from [14], one can obtain linear-space data structures to maintain a fair split tree, a well-separated pair decomposition, and the closest pair of a given point set in $\mathcal{O}(\log_B N)$ I/Os per point insertion or deletion. The details are straightforward. For the closest-pair problem, this fact has already been observed in [12]; but, by maintaining the well-separated pairs dynamically, one can obtain a direct externalization of the internal-memory data structure of [14], which is simpler than the solution proposed in [12].

8 Conclusions

We have provided I/O-efficient algorithms for constructing the well-separated pair decomposition of [15]. We have also demonstrated that, as in internal memory, the WSPD can be used to solve a number of proximity problems such as K-closest-pairs, K-nearest-neighbours, and the construction of t-spanners. While theoretically efficient, the algorithms are unlikely to be practical; the main obstacle being the large constant factors inherited from the construction of [15]. In particular, Callahan [11] provides an upper bound of $2((3s+6)\sqrt{d}+5)^d N$ on the number of pairs in the well-separated realization computed by procedure ComputeWSR. However, in internal memory, Narasimhan and Zachariasen [35] report that a WSPD-based algorithm for computing a geometric minimum spanning tree outperforms other approaches in dimensions $d \geq 3$ in practice. Instead of using the worst-case $\mathcal{O}(N \log N)$ time algorithm for constructing the fair split tree, they use the naïve construction, which splits the longest dimension in half, partitions the points around the partitioning hyperplane, and then recurses on the two subsets. They prove that the expected running time of this procedure is $\mathcal{O}(N \log N)$ and verify its performance experimentally. Their results suggest that, in practice, the size of the computed realization is nowhere near the upper bound provided by Callahan and that the naïve construction leads to practical algorithms. It would be interesting to verify experimentally whether the same is true in external memory.

Acknowledgements. We would like to thank Pankaj Agarwal, Lars Arge, and Pat Morin for helpful discussions. We would also like to thank the referees of our paper for excellent suggestions on how to improve the presentation of our results.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, September 1988.

[2] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–357. Kluwer Academic Publishers, 2002.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS, 1999.

[5] S. Arya, D. M. Mount, and M. Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 703–712, 1994.

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

[7] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 80–86, 1983.

[8] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23:214–229, 1980.

[9] S. N. Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete and Computational Geometry*, 19(2):175–195, 1998.

[10] P. B. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 332–341, 1993.

[11] P. B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1995.

[12] P. B. Callahan, M. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proceedings of the 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 1995.

[13] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1993.

[14] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 263–272, 1995.

[15] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest neighborsand n-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

[16] P. Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pages 169–177, 1986.

[17] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, January 1995.

[18] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 226–232, 1983.

[19] K. L. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the 19th ACM Symposium on the Theory of Computing*, pages 56–65, 1987.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, first edition, 1990.

[21] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry and Applications*, 11(3):305–338, 2001.

[22] M. T. Dickerson, R. L. S. Drysdale, and J.-R. Sack. Simple algorithms for enumerating interpoint distances and finding k nearest neighbors. *International Journal of Computational Geometry & Applications*, 2:221–239, 1993.

[23] M. T. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry: Theory and Applications*, 5:277–291, 1996.

[24] D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. North-Holland, 2000.

[25] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal of Algorithms*, 24:37–65, 1997.

[26] M. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. *SIAM Journal on Computing*, 27:1036–1072, 1998.

[27] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 714–723, November 1993.

[28] S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications. In *Proceedings of the 8th Annual European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pages 220–231. Springer-Verlag, September 2000.

[29] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete & Computational Geometry*, 7:13–28, 1992.

[30] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118:34–37, 1995.

[31] H.-P. Lenhof and M. Smid. Sequential and parallel algorithms for the k closest pairs problem. *International Journal of Computational Geometry & Applications*, 5:273–288, 1995.

[32] T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient batched range counting and its applications to proximity problems. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 244–255. Springer-Verlag, 2001.

[33] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies: Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[34] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge University Press, To appear.

[35] G. Narasimhan and M. Zachariasen. Geometric minimum spanning tree via well-separated pair decompositions. *Journal of Experimental Algorithmics*, 6, 2001.

[36] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, June/July 1993.

[37] W. Pugh. Skip lists: A probabilistic alternative to balanced search trees. *Communications of the ACM*, 33:668–676, 1990.

[38] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, 1976.

[39] J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, pages 207–210, 1991.

[40] J. S. Salowe. Enumerating interdistances in space. *International Journal of Computational Geometry & Applications*, 2:49–59, 1992.

[41] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.

[42] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, pages 37–67. Springer-Verlag, 1993.

[43] M. I. Shamos. Geometric complexity. In *Proceedings of the 7th Annual ACM Symposium on the Theory of Computing*, pages 224–233, 1975.

[44] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[45] M. Smid. Maintaining the minimal distance of a point set in less than linear time. *Algorithms Review*, 2:33–44, 1991.

[46] M. Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. North-Holland, 2000.

[47] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4:101–115, 1989.

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

[49] A. C. Yao. On constructing minimum spanning trees in k-dimensional space and related problems. *SIAM Journal on Computing*, 11:721–736, 1982.

[50] N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, School of Computer Science, Carleton University, 2002.