Cache-Oblivious Construction of a Well-Separated Pair Decomposition

Fabian Gieseke[*] Jan Vahrenhold[†]

Abstract

We present a cache-oblivious algorithm for computing a well-separated pair decomposition of a finite point set $S \subset \mathbb{R}^d$ using $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers.

1 Preliminaries

Given two point sets $A, B \subset \mathbb{R}^d$ with bounding hyperrectangles $R(A)$ and $R(B)$ and a real constant $s > 0$, we say that A and B are *well-separated* with respect to s if there are two disjoint balls $B_{(a,r)}$ and $B_{(b,r)}$ with radius $r \geq 0$ and centers $a, b \in \mathbb{R}^d$ such that $R(A) \subset B_{(a,r)}, R(B) \subset B_{(b,r)}$ and the distance between $B_{(a,r)}$ and $B_{(b,r)}$ is at least sr. We refer to s as the *separation ratio*. Callahan and Kosaraju [7] defined a *well-separated pair decomposition* (WSPD) for the point set S with separation ratio $s > 0$ as a sequence $\{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\}$ of pairs of nonempty subsets of S such that

1. $A_i \cap B_i = \emptyset$ for all $i = 1, \ldots, m$,
2. for each pair $\{p, q\}$ of distinct points of S, there is exactly one pair $\{A_i, B_i\}$ in the sequence, such that $p \in A_i$ and $q \in B_i$ or vice versa,
3. A_i and B_i are well-separated with respect to s for all $i = 1, \ldots, m$.

The integer m is called the *size* of the WSPD.

Observation 1 ([7]) *Let S be a set of points in $\mathbb{R}^d$ and let $s > 0$ be a real number. A WSPD for S with respect to s having size $\mathcal{O}(s^d|S|)$ can be computed in $\mathcal{O}(|S| \log |S| + s^d|S|)$ time.*

For the analysis in this paper we use the *cache-oblivious model* [8], which is a variant of the standard two-level *I/O model* introduced by Aggarwal and Vitter [1]. Their model defines N to denote the number of objects in the problem instance, $M \ll N$ to denote the number of objects fitting into internal memory, and $2 \leq B \leq M/2$ to denote the number of objects per disk block. In the *cache-oblivious model* of Frigo *et al.* [8], we also have the parameters N, M, and B for the analysis of an algorithm, but in the design-phase of an algorithm, the parameters of M and B must not be used. This allows us to model a multi-level hierarchical memory, where memory is transferred in blocks between any two adjacent layers of memory (and not only between memory and disk). Frigo *et al.* [8] showed that sorting N items requires $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ memory transfers while scanning N items can obviously be done in $\Theta(\frac{N}{B})$ memory transfers; both bounds match the I/O-bounds in the I/O-model. We will use the shorthands $\mathcal{O}(\mathrm{sort}(X)) = \mathcal{O}(\frac{X}{B} \log_{M/B} \frac{X}{B})$ and $\mathcal{O}(\mathrm{scan}(X)) = \mathcal{O}(\frac{X}{B})$.

2 Cache-Oblivious WSPD

We show that we can efficiently construct a WSPD in the cache-oblivious model of computation using only $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers. The algorithm follows the I/O-efficient approach of Govindarajan *et al.* [10]:

Observation 2 ([10]) *Given a set S of points in $\mathbb{R}^d$ and a real constant $s > 0$, a WSPD for S with separation ratio s and size $\mathcal{O}(s^d|S|)$ can be computed in $\mathcal{O}(\mathrm{sort}(|S|))$ I/Os using $\mathcal{O}(|S|/B)$ disk blocks.*

Most of their algorithm is based on scanning and sorting or on other techniques which have a cache-oblivious counterpart. However, some parts of their algorithm use the values of M and B (which are unknown to a cache-oblivious algorithm) or take advantage of data structures with no (direct) cache-oblivious counterpart. In particular, Govindarajan *et al.* use two such data structures, namely *buffer trees* [2] and *topology buffer trees* [12]. Replacing them by efficient cache-oblivious computation steps constitutes a major part of our modifications. We describe our cache-oblivious method by outlining the I/O-efficient algorithm along with the necessary modifications.[1]

In a nutshell, Govindarajan *et al.* [10] compute a WSPD for S as follows: They start by constructing a special binary tree T, called *fair split tree* of S, whose nodes are used to represent the pairs of the WSPD and whose leaves are in a one-to-one correspondence with the points in S. The idea is to build T recursively. First, a "partial" fair split tree T' with leaves of bounded *sizes*, i.e., whose leaves correspond to at most $|S|^\alpha$ points for some constant $1 - \frac{1}{2d} \leq \alpha < 1$, is constructed. They then recursively build the split trees for the leaves, proceeding with the optimal internal memory algorithm for every leaf whose size is at most M. After the computation of the split tree they

[*]Faculty of Computer Science, LS XI, TU Dortmund, 44227 Dortmund, Germany, `fabian.gieseke@cs.tu-dortmund.de`.

[†]Faculty of Computer Science, LS XI, TU Dortmund, 44227 Dortmund, Germany, `jan.vahrenhold@cs.tu-dortmund.de`

[1]See the thesis of Gieseke [9] for all details.

Algorithm 1 Construction of a fair split tree.

```
Function FairSplitTree($S, R_0$)
Require: A point set $S \subset \mathbb{R}^d$ and a box $R_0$ with $S \subset R_0$
Ensure: A fair split tree $T$ of $S$
1: if $|S| = 1$ then
2:     Construct a single-node fair split tree.
3: else
4:     $T' :=$ PartialFairSplitTree($S, R_0$).
       (Note: For each leaf $S_1, \dots, S_k$ of $T'$: $|S_i| \le |S|^\alpha$)
5:     for $i = 1$ to $k$ do
6:         $T_i :=$ FairSplitTree($S_i, \hat{R}(S_i)$).
           (Note: $\hat{R}(S_i)$ denotes an outer rectangle, which is box fulfilling certain properties)
7: return $T := T' \cup T_1 \cup \dots \cup T_k$
```

simulate the internal memory algorithm of Callahan and Kosaraju [7] for constructing the pairs $\{A_i, B_i\}$ in external memory using time-forward processing [2].

2.1 Construction of a Fair Split Tree

To compute T, we invoke FairSplitTree with the point set S and a centered minimal bounding cube R_0 containing S. The structure and the analysis of function FairSplitTree are close to those of the original I/O-efficient algorithm. However, we do not stop the recursion at leaves of size M (which are unknown to a cache-oblivious algorithm). Also, we set the constant α to $\alpha \in [1 - \frac{1}{4d}, 1)$, thus effectively slowing down the recursion. In Lemma 2 it will be shown that the function PartialFairSplitTree computes a valid partial fair split tree T' of S. Thus, the overall algorithm correctly computes the desired split tree T of S performing $\mathcal{I}(|S|) \le c \cdot \mathrm{sort}(|S|) + \sum_{i=1}^{k} \mathcal{I}(|S_i|) = \mathcal{O}(\mathrm{sort}(|S|))$ memory transfers. The linear bound for the space consumption follows from the fact that the function PartialFairSplitTree uses $\mathcal{O}(|S|/B)$ space and from the fact that $\sum_{i=1}^{k} |S_i| = |S|$.

Lemma 1 *Given a set S of points in $\mathbb{R}^d$, function* FairSplitTree *(Algorithm 1) computes a fair split tree of S in $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers using $\mathcal{O}(|S|/B)$ blocks of external memory.*

2.2 Construction of a Partial Fair Split Tree

It remains to show that PartialFairSplitTree (Algorithm 2) causes $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers and uses $\mathcal{O}(|S|/B)$ blocks of memory. The framework of this function is identical to that of the original I/O-efficient algorithm for computing the partial fair split tree T'. In the first step, a binary tree T_c, called *compressed pseudo split tree*, is computed whose nodes correspond to *boxes*, i.e. hyperrectangles R that fulfill the inequality $l_{\max}(R) \le 3 \cdot l_{\min}(R)$, where the latter notations denote the maximum and minimum length of R, respectively. In the second step, the tree T_c is expanded to a tree T'', called a *pseudo split tree.* In

Algorithm 2 Construction of a partial fair split tree.

```
Function PartialFairSplitTree($S, R_0$)
Require: A point set $S \subset \mathbb{R}^d$ and a box $R_0$ with $S \subset R_0$
Ensure: 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 $S$, where every node represents a box.
2: Expand $T_c$ to the pseudo split tree $T''$.
3: Remove every node of $T''$ that does not contain any points of $S$ and compress every path of nodes having one child into a single edge to obtain $T'$.
4: return $T'$
```

the last step, leaves which do not contain any points of S are removed and paths consisting of nodes having one child are compressed to single edges. The resulting tree is the desired partial fair split tree T' of S.

Govindarajan *et al.* first partition every dimension of the starting hyperrectangle R_0 into slabs each containing at most $|S|^\alpha$ points. For each dimension, these slabs are bounded by $\left\lceil |S|^{1-\alpha} \right\rceil + 1$ axis-parallel hyperplanes, called the *slab boundaries.* The two extreme slab boundaries in each dimension are positioned in such a way that they contain the two sides of the starting hyperrectangle R_0 in this dimension. The construction of the tree T_c is only based on the information provided by these slab boundaries and proceeds by splitting a hyperrectangle R fulfilling specific invariants into one or two smaller hyperrectangles fulfilling these invariants as well.

These invariants guarantee that each hyperrectangle side obtained by this splitting process can be described uniquely using a constant number of slab boundaries and a constant amount of additional information. As this is also true for the starting hyperrectangle R_0 (each of its sides is contained in a slab boundary), any hyperrectangle that can be obtained from R_0 through a sequence of splits can be described using a constant number of slab boundaries and a constant amount of extra information. The main idea of the I/O-efficient algorithm of Govindarajan *et al.* is to construct the set of *all* hyperrectangles that can be described this way. Callahan [5] calls these hyperrectangles *constructible* and bounds theirs total number:

Observation 3 ([5, 10]) *There are $\mathcal{O}(|S|^{2d(1-\alpha)})$ constructible hyperrectangles.*

Hence, by setting the value for the constant $\alpha \in [1 - \frac{1}{2d}, 1)$, Govindarajan *et al.* get $\mathcal{O}(|S|)$ constructible hyperrectangles. Note that, due to our minor, yet crucial modification of the value of α, we can bound the number of all constructible hyperrectangles by $\mathcal{O}(\sqrt{|S|})$. Ultimately, this will allow us to replace both the use of buffer trees as well as the use of topology buffer trees by efficient cache-oblivious computation steps. Govindarajan *et al.* then construct a graph Γ with nodes corresponding to the set of constructible hyperrectangles and with an edge set originating from the splitting process mentioned above. The tree T_c consists of all nodes of Γ that are accessible from R_0 along with their outgoing edges. The two main steps of the I/O-efficient construction of T_c are (a) building the graph Γ in a preprocessing step and (b) extracting the tree T_c afterwards.

Step 1a: Building $\mathbf{\Gamma}$ Constructing the slab boundaries is based on sorting and scanning, and a cache-oblivious implementation uses $\mathcal{O}(d \cdot \mathrm{sort}(|S|)) = \mathcal{O}(\mathrm{sort}(|S|))$ memory transfers. The vertex set of Γ, i.e. the set of all constructible hyperrectangles, can be constructed based on the information given by the slab boundaries. Govindarajan *et al.* do this by performing $\mathcal{O}(d)$ nested scans using $\mathcal{O}(\mathrm{scan}(|\Gamma|))$ I/Os each. They then use buffer trees [2] over the appropriate list of slab boundaries and answer I/O-efficient search queries on the constructed trees to augment every vertex R with a description of the largest hyperrectangle R' contained in R and bounded by slab boundaries as well as with a description of the slab boundary that comes closest to the "middle" of R in each dimension. They use this information to construct the edge set of Γ by a single scan of Γ's vertices. Thus, the construction of Γ uses $\mathcal{O}(\mathrm{sort}(|S|))$ I/Os.

The vertex set of Γ can be constructed in a cache-oblivious manner using $\mathcal{O}(d)$ nested scans as well. Instead of using buffer trees, we apply a brute-force approach for augmenting the nodes of Γ with the appropriate information. This approach simply scans the (sorted) list of slab boundaries in the i-th dimension to find the appropriate slab boundaries in each dimension. Due to our choice of α, each vertex of Γ can be processed this way using an overall number of $\mathcal{O}(d \cdot \mathrm{scan}(|S|^{1-\alpha})) = \mathcal{O}(\mathrm{scan}(\sqrt{|S|}))$ memory transfers. Thus, the overall process for all $\mathcal{O}(\sqrt{|S|})$ vertices of Γ requires $\mathcal{O}(\mathrm{scan}(|S|))$ memory transfers.

The final step, to construct the edge set of Γ, can be done analogously by a single scan over the vertex set of Γ taking $\mathcal{O}(\mathrm{scan}(|\Gamma|)) = \mathcal{O}(\mathrm{scan}(\sqrt{|S|}))$ memory transfers. This completes the construction of Γ.

Step 1b: Extracting $\mathbf{T_c}$ from $\mathbf{\Gamma}$ The I/O-efficient extraction of T_c from Γ can be implemented using time-forward processing and topological sorting. These steps can be performed efficiently in the cache-oblivious model [3, 4], and thus the extraction of T_c can be done the same way by spending $\mathcal{O}(\mathrm{sort}(|\Gamma|))$ memory transfers. With our choice of α, this step takes $\mathcal{O}(\mathrm{sort}(\sqrt{|S|}))$ memory transfers.

Step 2: Constructing $\mathbf{T''}$ Given the compressed pseudo split tree T_c for S, Govindarajan *et al.* construct the pseudo split tree T'' in three phases: In the first phase, they attach some leaves to T_c which were discarded during the construction of T_c. As at most one child is attached to each node of T_c, the resulting graph, called T_c^+, has size $\mathcal{O}(|T_c|)$ as well. This step can be implemented cache-obliviously in a straightforward manner (see [9] for the details) and takes $\mathcal{O}(\mathrm{scan}(\sqrt{|S|}))$ memory transfers.

To distribute the points of S to the proper boxes and regions, Govindarajan *et al.* answer so-called *deepest containment queries* on T_c^+ for all points in S using a topology buffer tree [12]. The distribution of all points takes $\mathcal{O}(\mathrm{sort}(|S|))$ I/Os.

Instead of using a topology buffer tree, we apply the following brute-force approach for distributing the points : Subdivide the space into cells bounded by the hyperplanes which occur as boundaries for the constructible hyperrectangles. Using the same argument as for bounding the number of constructible hyperrectangles ensures that there are only $\mathcal{O}(|S|^{2d(1-\alpha)}) = \mathcal{O}(\sqrt{|S|})$ cells of this type [5, 10]. Furthermore, a list of these cells can be constructed using $\mathcal{O}(d)$ nested scanning and sorting steps using $\mathcal{O}(\mathrm{sort}(\sqrt{|S|}))$ memory transfers. By sorting the points and the list of hyperplanes with respect to the ith coordinate and by subsequently scanning both sorted lists, we can label each point with the pair of adjacent hyperplanes in this dimension, which determines the slab that contains it. Processing all points and all d dimensions this way takes $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers and labels each point with a description of the cell that contains it. After computing this correspondence between the points in S and the cells, we compute an appropriate correspondence between the cells and the hyperrectangles and regions. By construction, each cell is either contained in a hyperrectangle being a leaf of T_c^+ or in a region $R \setminus C$, where (R, C) is a so-called *compressed edge* of T_c^+ [10]. As the number of cells and the number of hyperrectangles and regions are bounded by $\mathcal{O}(\sqrt{|S|})$, this correspondence can be obtained by a brute-force approach performing $\mathcal{O}(\mathrm{scan}(|S|))$ memory transfers. To compute the desired correspondence between the points in S and the hyperrectangles and regions, we first sort both lists lexicographically according to the cell entries. By scanning both sorted lists, we can subsequently augment each point with a representation of the hyperrectangle rather or region which contains the point. Both steps can be performed in $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers.

Finally, in the third phase, Govindarajan *et al.* replace every compressed edge by a sequence of splits which results in the desired tree T''. They prove that the resulting tree T'' has size $\mathcal{O}(|S|)$. As all techniques in this step are only based on scanning and sorting, the expansion of all edges can be performed cache-obliviously using $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers.

Step 3: Construction of $\mathbf{T'}$ Given the pseudo split tree T'' of S, Govindarajan *et al.* apply time-forward processing to remove every box R with $R \cap S = \emptyset$ from T'' and to compress all paths consisting of nodes having one child to a single edge in the resulting tree. The overall process is only based on scanning, sorting and time-forward processing and can therefore be implemented efficiently in the cache-oblivious model using $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers. The obtained tree is the desired fair split tree of S. As the bound for the space consumption follows from the layout of the algorithm, this completes the proof of the following lemma and, hence, also the proof of Lemma 1:

Lemma 2 *Given a set S of points in $\mathbb{R}^d$ and a constant $\alpha \in [1 - \frac{1}{4d}, 1)$, function* PartialFairSplitTree *(Algorithm 2) computes a partial fair split tree T' of S in $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers using $\mathcal{O}(|S|/B)$ blocks of external memory. Each leaf of T' represents a subset of S with at most $|S|^\alpha$ points.*

2.3 Construction of the Well-Separated Pairs

Once a fair split tree for the point set S is computed, Govindarajan *et al.* simulate the internal memory algorithm for constructing the pairs $\{A_i, B_i\}$ of the designated WSPD in external memory by applying the time-forward processing technique performing an overall number of $\mathcal{O}(\mathrm{sort}(|S|))$ I/Os. Besides time-forward processing, the I/O-efficient algorithm is only based on scanning and sorting. Thus, the complete procedure can be implemented in a cache-oblivious manner in $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers as well.

Theorem 3 *Given a set S of points in $\mathbb{R}^d$ and a real constant $s > 0$, a well-separated pair decomposition for S with separation ratio s having size $\mathcal{O}(s^d|S|)$ can be computed in $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers using linear space.*

3 Applications

Constructing t-spanners with a linear number of edges can be performed easily in a cache-oblivious manner using the WSPD. Given a set S of points in $\mathbb{R}^d$ and a real constant $t > 1$, construct a WSPD for S with separation ratio $4\frac{t+1}{t-1}$ having size $\mathcal{O}(|S|)$ and add, for each pair $\{A_i, B_i\}$ of the WSPD, exactly one (arbitrary) pair $\{x, y\}$ of points with $x \in A_i$ and $y \in B_i$ to an initially empty edge set E. Callahan and Kosaraju [6] proved that the resulting graph $G = (S, E)$ is a t-spanner for S with a linear number of edges.

Corollary 4 *Given a point set $S \subset \mathbb{R}^d$ and some constant $t > 1$, we can compute a t-spanner $G = (S, E)$ of linear size using $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers.*

Using Theorem 3, it is straightforward to restate [10, Theorem 4] in a cache-oblivious version:

Corollary 5 *Given a point set $S \subset \mathbb{R}^d$ and some constant $t > 1$, we can compute a t-spanner $G = (S, E)$ of linear size and diameter at most $2 \log |S|$ using $\mathcal{O}(\mathrm{sort}(|S|))$ memory transfers.*

In combination with our previous results for I/O-efficiently pruning dense spanners [11] we can prove:

Lemma 6 ([9, Theorem 6.4.8]) *Given a geometric graph $G = (S, E)$ which is a t-spanner for S for some constant $t > 1$ and given a constant $\varepsilon > 0$, we can compute a $(1 + \varepsilon)$-spanner $G' = (S, E')$ of G with $E' \subseteq E$ and $|E'| \in \mathcal{O}(|S|)$ using $\mathcal{O}(\mathrm{sort}(|E|))$ memory transfers.*

Acknowledgments

We thank Norbert Zeh for helpful discussions regarding the I/O-efficient WSPD algorithm [10].

References

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

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

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM J.Computing*, 36(6):1672–1695, 2007.

[4] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Intl. Symp. Algorithms and Computation*, volume 2518 of *LNCS*, pages 219–228. Springer, 2002.

[5] P. B. Callahan. *Dealing with higher dimensions: the well-separated pair decomposition and its applications.* Ph.D. thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, 1995.

[6] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. 4th Symp. Discrete Algorithms*, pages 291–300, 1993.

[7] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Symp. Foundations of Computer Science*, pages 285–299, 1999.

[9] F. Gieseke. Algorithmen zur Konstruktion und Ausdünnung von Spanner-Graphen im Cache-Oblivious-Modell. Technical Report TR07-3-005, Universität Dortmund, Fachbereich Informatik, July 2007. In German.

[10] S. Govindarajan, T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient well-separated pair decomposition and its applications. *Algorithmica*, 45(4):585–614, 2006.

[11] J. Gudmundsson and J. Vahrenhold. I/O-efficiently pruning dense spanners. In *Revised Selected Papers Japanese Conf. Discrete and Computational Geometry*, volume 3742 of *LNCS*, pages 106–116. Springer, 2005.

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