MAX-PLANCK-INSTITUT FÜR INFORMATIK

Sequential and parallel algorithms for the k closest pairs problem

Hans-Peter Lenhof Michiel Smid

MPI–I–92–134 August 1992

INFORMATIK

Im Stadtwald
66123 Saarbrücken
Germany

Sequential and parallel algorithms for the k closest pairs problem

Hans-Peter Lenhof Michiel Smid

MPI–I–92–134 August 1992

Sequential and parallel algorithms for the k closest pairs problem*

Hans-Peter Lenhof[†] Michiel Smid[†]

Max-Planck-Institut für Informatik

W-6600 Saarbrücken, Germany

August 7, 1992

Abstract

Let S be a set of n points in D-dimensional space, where D is a constant, and let k be an integer between 1 and $\binom{n}{2}$. A new and simpler proof is given of Salowe's theorem, i.e., a sequential algorithm is given that computes the k closest pairs in the set S in $O(n \log n + k)$ time, using $O(n + k)$ space. The algorithm fits in the algebraic decision tree model and is, therefore, optimal. Salowe's algorithm seems difficult to parallelize. A parallel version of our algorithm is given for the CRCW-PRAM model. This version runs in $O((\log n)^2 \log\log n)$ expected parallel time and has an $O(n \log n \log\log n + k)$ time-processor product.

1 Introduction

There has been a lot of interest in closest pair problems. In such problems, we are given a set of n points in D-dimensional space, and we want to compute the closest pair, all k closest pairs, or just the k-th closest pair. Distances are measured in an arbitrary L_t-metric, where $1 \leq t \leq \infty$. In this metric, the distance $d_t(p, q)$ between the points $p = (p_1, \ldots, p_D)$ and $q = (q_1, \ldots, q_D)$ is defined by

$$d_t(p,q) := \left(\sum_{i=1}^{D} |p_i - q_i|^t \right)^{1/t},$$

if $1 \leq t < \infty$, and for $t = \infty$, it is defined by

$$d_\infty(p,q) := \max_{1 \leq i \leq D} |p_i - q_i|.$$

*A preliminary version of this paper will appear in the Proceedings of the 33-rd Annual IEEE Symposium on Foundations of Computer Science, 1992, under the title *Enumerating the k closest pairs optimally*.

[†]This work was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

The problem of finding the closest pair has been solved already for a long time. Shamos and Hoey [13] and Bentley and Shamos [2] solve this problem, for the case $D = 2$ and $D \geq 2$, respectively, in $O(n \log n)$ time, which is optimal in the algebraic decision tree model. Recently, an optimal algorithm for the on-line version of this problem has been given by Schwarz et al. [12].

For the problem of computing the k-th closest pair, there are results by Agarwal et al. [1], who consider the L_t-metric for the planar case, and by Salowe [10], who shows how to select the k-th closest pair in the L_∞-metric in $O(n(\log n)^D)$ time, for any fixed $D \geq 2$.

For the problem of enumerating the k closest pairs, the first result was by Smid [14], who shows how to compute the $n^{2/3}$ closest pairs in a set of n points in D-space in $O(n \log n)$ time. This result was extended in two directions. First, for the planar case, Dickerson and Drysdale [5] compute the k closest pairs—ordered by their distances—in $O(n \log n + k \log n)$ time. Second, Salowe [11] gives an optimal algorithm that computes the k closest pairs in $O(n \log n + k)$ time. The latter result holds for an arbitrary, but fixed, dimension D. Salowe's algorithm uses complicated techniques such as parametric search and Vaidya's algorithm for the all-nearest-neighbors problem, see [15].

Katoh and Iwano [7] give a technique for solving related problems such as finding the k furthest pairs or the k closest/furthest bichromatic pairs. Their technique can also be applied to the k closest pairs problem. The result is an algorithm with running time $O(n \log n + k \log n \log(n^2/k))$. (See [8].)

In this paper, we give a new proof of Salowe's theorem: We show how to compute the k closest pairs in $O(n \log n + k)$ time. The algorithm works for any t, $1 \leq t \leq \infty$, any fixed dimension D and any k, $1 \leq k \leq \binom{n}{2}$. The algorithm fits in the algebraic decision tree model and is, therefore, optimal. (The constant factor is of the form $c^{D \log D}$, for some c.) We remark that our algorithm does not enumerate the k closest pairs in sorted order; the order can be arbitrary.

Comparing our technique with that of Salowe, our approach has the advantage that it is *completely self-contained*. Moreover, as we will see, our version can be parallelized, whereas this seems to be difficult for Salowe's algorithm.

Our algorithm is based on the fact that once the k-th smallest L_t-distance d_k is known, we can easily enumerate the k closest pairs: Consider a grid with cells of length d_k and distribute the points over the cells. Then, compare each point with all points that are contained in one of the 3^D neighboring cells. Of course, this algorithm takes too many pairs in consideration. All pairs that are looked at, however, have L_t-distance at most $2Dd_k$. (Equality can occur in the L_1-metric.) By a result of Salowe [11], see Lemma 2, the number of pairs considered is $O(k + n)$. Therefore, we find the k closest pairs in $O(n \log n + k)$ time.

Of course, the k-th smallest L_t-distance is not known at the start of the algorithm. We could approximate it by running Salowe's algorithm that finds the k-th smallest L_∞-distance. This takes, however, $O(n(\log n)^D)$ time. As we shall see, for our application, it suffices to approximate the k-th L_∞-distance. Such an approximation can be found in $O(n \log n)$ time.

We also give a parallel version of our algorithm. As mentioned already, Salowe's algorithm seems difficult to parallelize. Our parallel version uses the CRCW-PRAM model. It has an expected running time of $O((\log n)^2 \log\log n)$ and uses $n/\log n + k/((\log n)^2 \log\log n)$ processors. That is, the product of the time complexity and the number of processors is $O(n \log n \log\log n + k)$.

We have not found any results in the literature about the parallel complexity of the k closest pairs problem, except for the case $k = 1$. Cole and Goodrich [4] have shown that in the planar case, the closest pair in a set of n points can be found deterministically in $O(\log n)$ time using n processors. They obtain this result for the EREW-PRAM model. As far as we know, there are no non-trivial results for the closest pair problem in higher dimensions. Our algorithm finds the closest pair in a set of n points in D-space in $O((\log n)^2 \log\log n)$ expected time using $n/\log n$ processors. Therefore, for $D > 2$, this is the best result known at this moment.

The rest of this paper is organized as follows. In Section 2, we prove some combinatorial results that are needed in the analysis of the algorithm. In Section 3, we give the sequential algorithm, which consists of two phases. First, in Subsection 3.1, we compute an L_∞-distance with rank $\Theta(k+n)$. (Here, the term n appears because of a technical reason that will become clear later.) The algorithm implicitly manipulates all possible differences $|p_i - q_i|$, where $1 \leq i \leq D$ and p and q are points of the input set. It makes a binary search on these differences. (The way in which the binary search is controlled is due to Johnson and Mizoguchi [6], who use it to find the k-th smallest element in the Cartesian sum $X + Y$.) After a presorting step, which takes $O(n \log n)$ time, we can test in $O(n)$ time, if a difference $|p_i - q_i|$ approximates the desired L_∞-distance. The approximation is found within $O(\log n)$ iterations. In Subsection 3.2, this approximation is used to find the k closest pairs w.r.t. the L_t-metric. Of course, we can use the above sketched algorithm that uses a grid. Then, however, in order to distribute the points over the cells, we need the non-algebraic floor-function. Instead, we use a slightly degenerate grid that can be constructed without the floor-function.

In Section 4, we give a parallel implementation of our algorithm. We express the complexity of a parallel algorithm by giving the time bound $f(n)$ and the amount of *work* $g(n)$. This means that the algorithm takes $O(f(n))$ parallel time using $g(n)/f(n)$ processors. We finish the paper in Section 5 with some open problems.

In Subsection 3.1, we need to compute $\lfloor (l+h)/2 \rfloor$ for integers l and h in the range from 1 to n. At the start of the algorithm, we build an array

$$F[1:2n] = [0, 1, 1, 2, 2, 3, 3, \ldots, n-1, n-1, n].$$

Clearly, F can be constructed in $O(n)$ time using only algebraic functions. Then, the value of $\lfloor (l+h)/2 \rfloor$ is stored in $F[l+h]$ and, hence, can be retrieved in $O(1)$ time, without actually using the floor-function.

2 Some combinatorial results

We recall the notion of weighted medians. Let $x_1, x_2, \ldots, x_n$ be a sequence of real numbers such that every element x_i has a weight w_i, which is a positive real number. Let $W := \sum_{j=1}^{n} w_j$. Element x_i is called a *weighted median* if

$$\sum_{j: x_j < x_i} w_j < W/2 \quad \text{and} \quad \sum_{j: x_j \leq x_i} w_j \geq W/2. \tag{1}$$

The following lemma appears already in [6]. For completeness, we give a proof.

Lemma 1 *The weighted median of a set of n weighted real numbers can be computed in $O(n)$ time.*

Proof: The following algorithm finds the weighted median. In $O(n)$ time, compute the standard median, say x_i, and relabel the elements such that $x_j \leq x_i$ for $j \leq i$, and $x_j \geq x_i$ for $j \geq i$. Then compute the sums in (1). Using these sums we check if x_i is a weighted median. If so, we are finished. Otherwise, we know which half of the sequence contains the weighted median. We proceed recursively in this subsequence, which has length at most $n/2$. In this subsequence, we find an element with the appropriate weighted rank. ∎

We define a D-dimensional δ-cube as a hypercube of the form

$$[a_1 : a_1 + \delta) \times [a_2 : a_2 + \delta) \times \ldots \times [a_D : a_D + \delta),$$

where $a_1, a_2, \ldots, a_D$ are real numbers.

Definition 1 *Let S be a set of n points in D-space and let δ be a positive real number. A collection R of D-dimensional δ-cubes is called a* δ-covering *of S, if*

1. *the cubes are pairwise disjoint,*
2. *each cube contains at least one point of S,*
3. *each point in S is contained in one cube.*

Let R be a δ-covering of S. Label the cubes (arbitrarily) $1, 2, \ldots, r := |R|$ and define n_i to be the number of points of S that are contained in the i-th cube of R. We define

$$\Sigma(S, R) := \sum_{i=1}^{r} \binom{n_i}{2}.$$

If δ is a real number, then we denote by $r_\infty(\delta)$ the number of L_∞-distances in S that are less than δ.

The following lemmas will be used throughout the rest of the paper.

Lemma 2 (Salowe [11]) *Let S be a set of n points in D-space and let δ be a positive real number. Then,*

$$r_\infty(2\delta) \leq 5^D \left(r_\infty(\delta) + n\right).$$

After Lemma 3, we indicate how Lemma 2 can be proved.

Lemma 3 *Let S be a set of n points in D-space, let δ be a positive real number, and let R be a δ-covering of S. Then,*

$$\Sigma(S, R) \leq r_\infty(\delta) \leq 4^D(\Sigma(S, R) + n).$$

Proof: Clearly, $\binom{n_i}{2}$ counts the number of pairs of points that are contained in the i-th cube of R. Each such pair has L_∞-distance less than δ. This proves the left inequality. To prove the other inequality, consider a cube h in R:

$$h = [a_1 : a_1 + \delta) \times \ldots \times [a_D : a_D + \delta).$$

Let h' be the 3δ-cube

$$h' = [a_1 - \delta : a_1 + 2\delta) \times \ldots \times [a_D - \delta : a_D + 2\delta), \tag{2}$$

i.e., the cube with sides of length 3δ that contains h in its center. This cube h' intersects at most 4^D cubes of R: This follows from the fact that an interval of length 3δ can intersect at most 4 intervals from a set of pairwise disjoint intervals of length δ.

We call the cubes of R that intersect h' the *neighboring cubes* of h. Let C_i denote the set of labels of the neighboring cubes of the i-th cube in R. Note that the relation "neighboring cube" is reflexive and symmetric, i.e.,

$$i \in C_i \quad \text{and} \quad i \in C_j \text{ iff } j \in C_i.$$

Now consider two points $p \neq q$ in S having L_∞-distance less than δ. Then either p and q are contained in the same cube of R, or p is contained in a cube, say $h \in R$, and q is contained in a neighboring cube of h. It follows that

$$r_\infty(\delta) \leq \frac{1}{2} \sum_{i=1}^{r} n_i \sum_{j \in C_i} n_j.$$

To prove the right inequality, it suffices to show that this double summation is bounded above by $4^D(\Sigma(S, R) + n)$. This follows from some elementary calculations:

$$\begin{aligned}
\sum_{i=1}^{r} n_i \sum_{j \in C_i} n_j &\leq \sum_{i=1}^{r} \sum_{j \in C_i} (\max(n_i, n_j))^2 \\
&= \sum_{u=1}^{r} |\{v : u \in C_v, \max(n_u, n_v) = n_u\}| \times n_u^2 \\
&\leq \sum_{u=1}^{r} |\{v : u \in C_v\}| \times n_u^2 \\
&= \sum_{u=1}^{r} |\{v : v \in C_u\}| \times n_u^2 \\
&= \sum_{u=1}^{r} |C_u| \times n_u^2 \\
&\leq 4^D \sum_{u=1}^{r} n_u^2.
\end{aligned}$$

Using the fact that $n_u^2 = n_u + 2\binom{n_u}{2}$, the proof can easily be completed. ∎

Remark: Salowe has a constant of 5^{2D} instead of 5^D. Lemma 2 can be proved in a similar way as Lemma 3, by taking for R the non-empty δ-cubes of a grid with cells of length δ, and taking for h' in (2) a 5δ-cube. Counting all pairs with L_∞-distance less than 2δ gives the bound in Lemma 2. In fact, Lemma 3 and its proof are basically the same as Salowe's proof of Lemma 2, see [11].

We give two corollaries. The constants that appear are rather large. By a more careful analysis, however, they can be decreased, but they remain exponential in D. Since such an analysis does not give more insight into our algorithm, we do not give it here.

Corollary 1 *Let S be a set of n points in D-space and let δ be a positive real number. Let R be any δ-covering of S, such that $k \leq \Sigma(S,R) \leq 20^D(k+2n)$. Then,*

$$k \leq r_\infty(\delta) \leq 80^D(k+3n).$$

Proof: This follows immediately from Lemma 3. ∎

Corollary 2 *Let S be a set of n points in D-space and let $\delta^\star$ be the $20^D(k+2n)$-th smallest L_∞-distance in S. Let R be any $\delta^\star$-covering of S. Then,*

$$k \leq \Sigma(S,R) \leq 20^D(k+2n).$$

Proof: We know from Lemma 3 that

$$\Sigma(S,R) \leq r_\infty(\delta^\star) \leq 4^D(\Sigma(S,R)+n).$$

Since $r_\infty(\delta^\star) < 20^D(k+2n)$, it follows that $\Sigma(S,R) \leq 20^D(k+2n)$. From Lemma 2, we know that $r_\infty(2\delta^\star) \leq 5^D(r_\infty(\delta^\star)+n)$. Since $r_\infty(2\delta^\star) \geq 20^D(k+2n)$, we get

$$20^D(k+2n) \leq 5^D(r_\infty(\delta^\star)+n) \leq 20^D(\Sigma(S,R)+2n). \tag{3}$$

Therefore, $\Sigma(S,R) \geq k$. ∎

Remark: In this proof, we need a lower bound on $r_\infty(\delta^\star)$. Since many L_∞-distances may be equal, the definition of $\delta^\star$ does not immediately give us such a lower bound. We derive it using $r_\infty(2\delta^\star)$.

3 The sequential k closest pairs algorithm

Throughout this section, S is a set of n points in D-space. We assume that k is such that $20^D(k+2n) \leq \binom{n}{2}$. If this is not the case, we can find the k closest pairs, by considering all pairs of points and selecting the k that are closest. This takes $O(n^2) = O(k)$ time, which is clearly optimal.

The algorithm consists of two phases. In the first phase, we search for a real number δ for which the condition of Corollary 1 holds. (By Corollary 2, such a δ exists.) Then, in the second phase, we use this δ to enumerate all L_∞-distances that are less than $D\delta$. There are $O(k+n)$ such distances. We extract from them the k smallest L_t-distances.

3.1 The approximation phase

We want to find an L_∞-distance δ in the set S, such that $r_\infty(\delta)$ lies in between k and $80^D(k+3n)$. There are two points $p = (p_1, \ldots, p_D)$ and $q = (q_1, \ldots, q_D)$ in S, and an i, such that $|p_i - q_i| = \delta$. In order to find this δ, we do a binary search on all possible differences $|p_i - q_i|$. Of course, we maintain the candidate differences in an implicit way. (This technique appears already in [6].) Note that we do not search for the difference $|p_i - q_i|$ with a certain rank; we search for the L_∞-distance with this rank.

We maintain the following information:

1. Arrays $A_1, \ldots, A_D$ of length n, where A_i contains the points of S sorted w.r.t. their i-th coordinates. For each $1 < i \leq D$, each point in A_i contains a pointer to its copy in A_{i-1}.
2. For each $1 \leq i \leq D$ and $1 \leq j < n$, we store with $A_i[j]$ an interval $[l_{ij} : h_{ij}]$, where l_{ij} and h_{ij} are integers, such that $j < l_{ij} \leq h_{ij} + 1 \leq n + 1$.

We define the set of *candidate differences* as follows. Let $p = (p_1, \ldots, p_D)$ and $q = (q_1, \ldots, q_D)$ be two distinct points in S, and let $1 \leq i \leq D$. Moreover, let j and j' be such that $A_i[j] = p$ and $A_i[j'] = q$. Assume w.l.o.g. that $j < j'$. Then $|q_i - p_i|$ is a candidate difference iff $l_{ij} \leq j' \leq h_{ij}$. Hence, the total number of candidate differences is equal to

$$\sum_{i=1}^{D} \sum_{j=1}^{n-1} (h_{ij} - l_{ij} + 1).$$

The algorithm makes a sequence of iterations. At each iteration, this summation is decreased by a factor of at least one fourth. We maintain the following

Invariant: At each moment, the $20^D(k+2n)$-th smallest L_∞-distance δ^* is contained in the set of candidate differences.

Initialization: At the start of the algorithm, we build the arrays $A_1, \ldots, A_D$ and add the pointers between them. Then, for each $1 \leq i \leq D$ and $1 \leq j < n$, we store with $A_i[j]$ the interval $[l_{ij} : h_{ij}] = [j+1 : n]$.

Now, the algorithm starts with the

Iteration:

Step 1. For each $1 \leq i \leq D$ and $1 \leq j < n$, such that $l_{ij} \leq h_{ij}$, take the pair

$$A_i[\lfloor (l_{ij} + h_{ij})/2 \rfloor] \quad \text{and} \quad A_i[j]$$

and take the (positive) difference of their i-th coordinates. Give this difference *weight* $h_{ij} - l_{ij} + 1$. This gives a sequence of at most $D(n-1)$ weighted differences.

Step 2. Compute a weighted median δ of these weighted differences.

Step 3. Construct a δ-covering R of S, and compute $\Sigma(S, R)$. There are three possible cases.

1. If $k \leq \Sigma(S, R) \leq 20^D(k + 2n)$, then output δ and stop.
2. If $\Sigma(S, R) < k$, then for each pair
$$A_i[\lfloor (l_{ij} + h_{ij})/2 \rfloor] \quad \text{and} \quad A_i[j]$$
selected in the first step such that the difference of their i-th coordinates is at most δ, set $l_{ij} := \lfloor (l_{ij} + h_{ij})/2 \rfloor + 1$. Go to Step 1.
3. If $\Sigma(S, R) > 20^D(k + 2n)$, then for each pair
$$A_i[\lfloor (l_{ij} + h_{ij})/2 \rfloor] \quad \text{and} \quad A_i[j]$$
selected in the first step such that the difference of their i-th coordinates is at least δ, set $h_{ij} := \lfloor (l_{ij} + h_{ij})/2 \rfloor - 1$. Go to Step 1.

The construction of the δ-covering R will be described later. First we prove two lemmas.

Lemma 4 *The algorithm correctly maintains the invariant.*

Proof: After the initialization, the total number of candidate differences is equal to
$$\sum_{i=1}^{D} \sum_{j=1}^{n-1} (n - j) = D\binom{n}{2},$$
i.e., the set of candidate differences equals the set of all $D\binom{n}{2}$ differences $|p_i - q_i|$. Therefore, the invariant holds initially. Consider one iteration. First assume that Case 2 of Step 3 applies, i.e., $\Sigma(S, R) < k$. Then, by Lemma 3,
$$r_\infty(\delta) \leq 4^D(\Sigma(S, R) + n) < 4^D(k + n).$$
We know from (3) that
$$r_\infty(\delta^*) \geq 4^D(k + 2n) - n > 4^D(k + n).$$
Therefore, $r_\infty(\delta) < r_\infty(\delta^*)$ and, hence, $\delta < \delta^*$. The algorithm only removes differences $|p_u - q_u|$ from the set of candidate differences that are at most equal to δ. Hence, at the end of the iteration, the invariant still holds.

If Case 3 of Step 3 applies, then we know from Lemma 3 that
$$r_\infty(\delta) \geq \Sigma(S, R) > 20^D(k + 2n).$$
Since $r_\infty(\delta^*) < 20^D(k + 2n)$, we infer that $\delta > \delta^*$. Hence, we can remove differences $|p_u - q_u|$ from the set of candidate differences that are at least equal to δ, without invalidating the invariant. ∎

Lemma 5 *The algorithm makes at most* $\log_{4/3}(D\,n^2) = O(\log n)$ *iterations.*

Proof: Let W (resp. W') be the total number of candidate differences at the start of (resp. immediately after) an iteration. Moreover, let l_{ij} and h_{ij} (resp. l'_{ij} and h'_{ij}) denote the endpoints of the intervals at the start of (resp. immediately after) this iteration.

Suppose that Case 2 of Step 3 applies. If $l'_{ij} \neq l_{ij}$, then $l'_{ij} = \lfloor (l_{ij} + h_{ij})/2 \rfloor + 1$ and $h'_{ij} = h_{ij}$. Since l_{ij} and h_{ij} are integers, we have $l'_{ij} \geq (l_{ij} + h_{ij} + 1)/2$. Therefore,

$$\begin{aligned} W &= W' + \sum_{i,j:l'_{ij} \neq l_{ij}} (l'_{ij} - l_{ij}) \\ &\geq W' + \frac{1}{2} \sum_{i,j:l'_{ij} \neq l_{ij}} (h_{ij} - l_{ij} + 1) \\ &\geq W' + \frac{1}{2}\frac{W}{2}, \end{aligned}$$

where the last inequality follows from the fact that δ is a weighted median. Hence,

$$W' \leq \frac{3}{4} W. \tag{4}$$

If Case 3 of Step 3 applies, Equation (4) can be proved in the same way.

The algorithm terminates as soon as it finds a real number δ such that $k \leq \Sigma(S,R) \leq 20^D(k + 2n)$ holds for the corresponding δ-covering R. By the invariant and Corollary 2, such a δ is always contained in the set of candidate differences. Since this set gets smaller in each iteration, the algorithm must find such a δ. This proves that the algorithm terminates.

Let z be the number of iterations made by the algorithm. It follows from the invariant that the set of candidate differences is never empty. Hence, if we denote the number of candidate differences after z iterations by W_z, then $W_z \geq 1$. Since there are $D\binom{n}{2}$ candidate differences before the first iteration, it follows from (4) that

$$W_z \leq \left(\frac{3}{4}\right)^z D \binom{n}{2}.$$

Therefore,

$$z \log(4/3) \leq \log\left(D\binom{n}{2}\right).$$

This proves the lemma. ∎

It remains to give an algorithm that constructs a δ-covering R. Of course, we can take a D-dimensional grid of side lengths δ and distribute the points over the cells. Then, however, we must use the floor-function and, hence, the algorithm falls outside the algebraic decision tree model. We give a recursive algorithm that computes a δ-covering using only algebraic functions.

Constructing a δ-covering:

Step 1. The algorithm walks along the array A_1. Let a_1 be the first coordinate of the point $A_1[1]$. Let $i \geq 1$, and assume that $a_1, \ldots, a_i$ are defined already.

If there is a point in S having a first coordinate lying in the half-open interval $[a_i : a_i + \delta)$, then we set $a_{i+1} := a_i + \delta$. Otherwise, we set a_{i+1} to the value of the first coordinate of the first point in A_1 that lies "to the right" of a_i. If we have reached the end of the array A_1, we set $a_{i+1} := a_i + \delta$ and the construction of the a_j's stops.

This gives a sequence of δ-intervals $[a_1 : a_1 + \delta), [a_2 : a_2 + \delta), \ldots, [a_l : a_l + \delta)$, for some l. During the walk along A_1, we give each point a pointer to the δ-interval to which its first coordinate belongs.

If $D = 1$, the algorithm is finished. Otherwise, $D > 1$, and the algorithm proceeds with Step 2:

Step 2. We partition S into subsets $S_1, \ldots, S_l$, as follows: Walk along the array A_2. For each point p encountered, follow the pointer to its copy in A_1, and follow the pointer stored there to the interval, say $[a_i : a_i + \delta)$, to which the first coordinate of p belongs. We add point p to subset S_i; more precisely, we store p at the end of a list representing S_i. At the end, we have l lists, where the i-th one stores the points of S_i sorted by their 2-nd coordinates.

For $i = 1, \ldots, l$, do the following. Use the same algorithm recursively to compute a δ-covering for the set S_i, where we take only the last $D-1$ coordinates into account. This gives a collection of $(D-1)$-dimensional δ-cubes of the form

$$[b_2 : b_2 + \delta) \times [b_3 : b_3 + \delta) \times \ldots \times [b_D : b_D + \delta),$$

together with a corresponding partition of S_i. Replace each such cube by the D-dimensional δ-cube

$$[a_i : a_i + \delta) \times [b_2 : b_2 + \delta) \times [b_3 : b_3 + \delta) \times \ldots \times [b_D : b_D + \delta).$$

The resulting cubes—for all i together—form the desired δ-covering R of S.

It is clear that once the δ-covering R has been constructed, we can compute $\Sigma(S, R)$ in $O(n)$ time. In fact, this value can be computed during the construction of R.

Lemma 6 *Let R be the set of hypercubes that are computed by the above algorithm. Then, R is a δ-covering of S. Moreover, the algorithm computes R and $\Sigma(S, R)$ in $O(n)$ time.*

Proof: It is clear that R is a δ-covering of S. Let $T(n, D)$ denote the running time of the algorithm. Since at the start of a recursive call, the points are sorted already by the appropriate coordinates, we have

$$\begin{aligned} T(n,1) &= O(n), \\ T(n,D) &= O(n) + \sum_{i=1}^{l} T(n_i, D-1), \text{ if } D \geq 2, \end{aligned}$$

for integers $n_i \geq 1$ such that $\sum_{i=1}^{l} n_i = n$. Using induction, it follows that $T(n, D) = O(n)$, because D is a constant. ∎

Theorem 1 *In $O(n \log n)$ time and using $O(n)$ space, we can compute a real number δ, such that $k \leq r_\infty(\delta) \leq 80^D(k+3n)$.*

Proof: The algorithm outputs a real number δ such that $k \leq \Sigma(S,R) \leq 20^D(k+2n)$ holds for the corresponding δ-covering R. Then, Corollary 1 implies the bounds on $r_\infty(\delta)$.

The initialization of the algorithm takes $O(n \log n)$ time. Moreover, by Lemmas 1, 5 and 6, $O(\log n)$ iterations are made, each taking $O(n)$ time. This proves that the entire algorithm has running time $O(n \log n)$. Finally, it is clear that the algorithm uses only linear space. ∎

3.2 The enumeration phase

At this moment, we have found a real number δ, such that $k \leq r_\infty(\delta) \leq 80^D(k+3n)$. That is, the number of L_∞-distances in S that are less than δ lies in between k and $80^D(k+3n)$.

In the enumeration phase, we find all L_t-distances that are less than $D\delta$. From these distances, we extract the k smallest ones. The details are as follows. (We use the notion of neighboring cube, which was defined in the proof of Lemma 3.)

Step 1. Construct a $D\delta$-covering R of S. Note that the algorithm outputs the cubes of R in lexicographical order.

Step 2. Build a list storing the following pairs of points of S: For each $D\delta$-cube h in R, all pairs (p,q), $p \neq q$, where $p \in h$ and q is contained in some neighboring cube of h that is (lexicographically) at least equal to h. (In this way, we get each pair only once.) These neighboring cubes can be found as follows. Let

$$h = [a_1 : a_1 + D\delta) \times \ldots \times [a_D : a_D + D\delta).$$

Search for all cubes in R that contain any of the 4^D points

$$(a_1 + \epsilon_1 D\delta, \ldots, a_D + \epsilon_D D\delta),$$

where $\epsilon_1, \ldots, \epsilon_D \in \{-1, 0, 1, 2\}$, and that are lexicographically at least equal to h.

Step 3. Take the list of pairs that results from the previous step and find the k closest pairs w.r.t. the L_t-distance.

Lemma 7 *Given δ, the algorithm finds the k closest pairs in the set S in $O(n \log n + k)$ time, using $O(n+k)$ space.*

Proof: Let d_t^k (resp. d_∞^k) denote the k-th smallest L_t-distance (resp. L_∞-distance) in S. Since $r_\infty(\delta) \geq k$, we have $d_\infty^k < \delta$. Moreover, since $d_t(p,q) \leq D\, d_\infty(p,q)$ for $1 \leq t \leq \infty$, we have $d_t^k \leq D\, d_\infty^k$. Hence, $d_t^k < D\delta$. In Step 2, all pairs (p,q), $p \neq q$, such that $d_t(p,q) < D\delta$, are found. Hence, in Step 2, all k L_t-closest pairs are added to the list. This proves the correctness of the algorithm.

Considering the time complexity, Step 1 takes $O(n)$ time. All pairs that are found in Step 2 have L_∞-distance less than $3D\delta$. By repeated application of Lemma 2, it follows that the number of pairs found in this step is at most

$$r_\infty(3D\delta) \leq 5^{D\lceil\log 3D\rceil} r_\infty(\delta) + \frac{5^{D(1+\lceil\log 3D\rceil)} - 5^D}{5^D - 1} n = O(k+n).$$

Moreover, in the second step, we make at most $4^D n$ point location queries. Each query can be solved in $O(\log n)$ time by a binary search. Hence, the total time for Step 2 is bounded by $O(n \log n + r_\infty(3D\delta)) = O(n \log n + k)$.

For Step 3, we use a linear time algorithm to find the k-th smallest L_t-distance d_t^k. Then, by making one scan over the list, we extract all pairs having an L_t-distance less than or equal to d_t^k. The total time for Step 3 is bounded by the size of the list, i.e., $O(r_\infty(3D\delta)) = O(k+n)$.

The algorithm uses an amount of space that is bounded by $O(n + r_\infty(3D\delta)) = O(n+k)$. ■

This completes the description of the algorithm and its analysis. Combining Theorem 1 and Lemma 7, we get the main result of this paper:

Theorem 2 *Let S be a set of n points in D-space and let $1 \leq k \leq \binom{n}{2}$. We can find the k closest pairs (w.r.t. the L_t-metric) in the set S in $O(n \log n + k)$ time, using $O(n+k)$ space, which is optimal.*

4 A parallel implementation

In this section, we parallelize the k closest pairs algorithm. Let S be a set of n points in D-space and let k be an integer between 1 and $\binom{n}{2}$. First assume that $20^D(k+2n) > \binom{n}{2}$. Then we apply Cole's optimal selection algorithm of [3]. This algorithm runs on a CRCW-PRAM and finds the k-th smallest in a set of size m, in $O(\log m \log^* m / \log\log m)$ time with $O(m)$ work. In our case, $m = \binom{n}{2}$. We find the k-th smallest L_t-distance d_t^k, in $O((\log n)^2 \log\log n)$ time with $O(n^2)$ work, by decreasing the number of processors in Cole's algorithm. Then, within the same bounds, we select all k L_t-distances that are at most equal to d_t^k. Clearly, this is an optimal algorithm.

From now on, we assume that $20^D(k+2n) \leq \binom{n}{2}$. The basic strategy of the algorithm is the same as before. We treat the two phases separately.

4.1 The approximation phase

As in Subsection 3.1, we search for an L_∞-distance δ, such that $r_\infty(\delta)$ lies in between k and $80^D(k+3n)$. We maintain the following information:

1. Arrays $A_1, \ldots, A_D$ of length n, where A_i contains the points of S sorted w.r.t. their i-th coordinates.

2. For each $1 \leq i \leq D$ and $1 \leq j < n$, we store with $A_i[j]$ an interval $[l_{ij} : h_{ij}]$, where l_{ij} and h_{ij} are integers, such that $j < l_{ij} \leq h_{ij} + 1 \leq n + 1$.

The set of *candidate differences* is defined as before. Moreover, we maintain the same invariant as in the sequential algorithm. We go through the algorithm of Subsection 3.1 and show how to parallelize each of the steps.

It is clear how the initialization of the algorithm can be performed by a CRCW-PRAM in $O((\log n)^2)$ time and $O(n \log n)$ work.

Consider one iteration. It is clear how to perform Step 1 in $O(\log n)$ time and $O(n)$ work. For Step 2, we have to be more careful, because it is not clear how a weighted median can be computed in $O(\log n)$ time and $O(n)$ work. In the previous section, the number of candidate differences was decreased by a factor of at least one fourth. We take the following approach, which will decrease this number by a factor of at least one eighth:

Step 2. Let $s_1, s_2, \ldots, s_m$ be the sequence of differences obtained from Step 1. Note that $m \leq D(n-1)$. Let $w(s_i) := h_{ij} - l_{ij} + 1$ denote the weight of s_i.

For each $0 \leq i \leq m/\lceil \log n \rceil$, one processor uses a sequential algorithm to compute a weighted median b_i of the subsequence

$$s_{1+i\lceil \log n \rceil}, s_{2+i\lceil \log n \rceil}, \ldots, s_{(i+1)\lceil \log n \rceil}. \tag{5}$$

This gives a sequence $b_0, b_1, \ldots, b_l$ of real numbers, where $l = 1 + m/\lceil \log n \rceil = O(n/\log n)$. We give each b_i weight

$$w'(b_i) := \frac{1}{2} \sum_{j=1+i\lceil \log n \rceil}^{(i+1)\lceil \log n \rceil} w(s_j).$$

In $O(\log n)$ time and $O(n)$ work, we sort the b_i's. Then, we use Equation (1) and a prefix-sum algorithm to compute a weighted median of the b_i's. This also takes $O(\log n)$ time and $O(n)$ work. Let δ denote this weighted median. This concludes the parallel implementation of Step 2.

Except for the construction of the δ-covering R of S and the computation of $\Sigma(S, R)$, it is clear how to perform Step 3 in $O(\log n)$ time and $O(n)$ work.

As in Subsection 3.1, we first give two lemmas. Then we consider the problem of computing $\Sigma(S, R)$.

Lemma 8 *The algorithm correctly maintains the invariant.*

Proof: The proof is exactly the same as that of Lemma 4. ∎

Lemma 9 *The algorithm makes at most* $\log_{8/7}(D\,n^2) = O(\log n)$ *iterations.*

Proof: Let W (resp. W') be the total number of candidate differences at the start of (resp. immediately after) an iteration. Suppose that Case 2 of Step 3 applies, i.e., $\Sigma(S, R) < k$. We discard at least

$$\sum_{i : b_i \leq \delta} \quad \sum_{\substack{j \,:\, s_j \leq b_i \,\wedge \\ 1 + i\lceil \log n \rceil \leq j \leq (i+1)\lceil \log n \rceil}} \quad \frac{1}{2} w(s_j)$$

elements from the set of candidate differences. Since b_i is a weighted median of the sequence (5), the inner summation is at least

$$\frac{1}{2} \sum_{j=1+i\lceil \log n \rceil}^{(i+1)\lceil \log n \rceil} \frac{1}{2}\, w(s_j) = \frac{1}{2}\, w'(b_i).$$

Therefore, the number of candidate differences discarded is at least

$$\frac{1}{2} \sum_{i: b_i \leq \delta} w'(b_i).$$

Since δ is a weighted median of the b_i's, the latter summation is at least

$$\frac{1}{4} \sum_{i \geq 0} w'(b_i) = \frac{1}{8} \sum_{j=1}^{m} w(s_j) = \frac{1}{8}\, W.$$

It follows that $W' \leq 7/8\, W$. If Case 3 of Step 3 applies, this inequality can be proved in the same way. The proof can be completed as in Lemma 5. ∎

It remains to give a parallel algorithm for constructing a δ-covering R and computing $\Sigma(S, R)$.

Constructing a δ-covering: The algorithm repeats Steps 1-4 below for each dimension m, where $1 \leq m \leq D$. For convenience, we set $A_m[0] := -\infty$.

Step 1. For each $0 \leq i \leq n/\log n$, the i-th processor does the following: For $j = i\lceil \log n \rceil, 1 + i\lceil \log n \rceil, \ldots, (i+1)\lceil \log n \rceil - 1$, if $A_m[j+1] \geq A_m[j] + \delta$, then mark element $A_m[j+1]$.

Note that $A_m[1]$ is always marked. For each marked element $A_m[j+1]$, there will be at least one δ-cube in the final δ-covering that starts its interval at dimension m at $A_m[j+1]$. Clearly, this step takes $O(\log n)$ time and $O(n)$ work.

Step 2. We build a tree storing the elements of the array A_m in sorted order in its leaves. Each node v of this tree contains a pointer to the rightmost element in its subtree that is marked. If there are no marked elements in the subtree of v, this pointer contains the value *nil*.

Since the array A_m is sorted already, this step takes $O(\log n)$ time and $O(n)$ work.

Step 3. For each $0 \leq i \leq n/\log n$, one processor does the following: Let p_m be the m-th coordinate of the point $p := A_m[1 + i\lceil \log n \rceil]$. If p is marked, nothing is done: We know already the left endpoint of the m-th interval of the δ-cube that contains p.

Assume that p is unmarked. Using the tree constructed in Step 2, find the rightmost element q in the array A_m that is to the left of p and that is marked. (Note that q exists. Moreover, a δ-cube starts its m-th interval at q.) Let

$$c := q_m + \delta \lfloor (p_m - q_m)/\delta \rfloor,$$

where q_m is the m-th coordinate of q. Then, $c \leq p_m < c + \delta$. The δ-cube for p in the final δ-covering will have c as left endpoint of its m-th interval.

How do we compute $\lfloor (p_m - q_m)/\delta \rfloor$? We know that all elements in A_m that are between q and p are unmarked. That is, the difference of the m-th coordinates of two consecutive elements between q and p is less than δ. Therefore, $0 < p_m - q_m < n\delta$, which implies that $0 \leq \lfloor (p_m - q_m)/\delta \rfloor < n$. We compute this integer in $O(\log n)$ time by binary search. In this way, we only use algebraic functions.

At this moment, the i-th processor knows the left endpoint of the m-th interval of the δ-cube to which $p = A_m[1 + i\lceil \log n \rceil]$ belongs. This processor uses Step 1 of the sequential algorithm of Subsection 3.1—for dimension m—to find δ-intervals for the points $A_m[1 + i\lceil \log n \rceil], A_m[2 + i\lceil \log n \rceil], \ldots, A_m[(i+1)\lceil \log n \rceil]$.

Step 3 takes $O(\log n)$ time and $O(n)$ work. At the end of this step, we know for each point the left boundary of the m-th interval of its δ-cube. Moreover, since A_m is sorted, we have these left boundaries in sorted order.

Step 4. We rank the n left boundaries obtained from the previous step. (Equal boundaries get the same rank.) This takes $O(\log n)$ time and $O(n)$ work.

At the end of this step, we have for each point the number of its δ-slab for the m-th dimension. (These slabs are numbered starting at one.)

Step 5. Note that we have carried out the first four steps for each dimension. Therefore, at this moment, each point has stored with it two vectors of length D. If point p has vectors $(c_1, c_2, \ldots, c_D)$ and $(k_1, k_2, \ldots, k_D)$, then p is contained in the δ-cube with lower-left corner $(c_1, c_2, \ldots, c_D)$. This δ-cube is part of the k_m-th δ-slab along the m-th axis.

These vectors implicitly define the δ-covering R. So, it remains to compute $\Sigma(S, R)$. Note that each k_m is an integer in the range from 1 to n. We consider each vector $(k_1, k_2, \ldots, k_D)$ as an integer between 1 and n^{D+1}. Then, using the randomized algorithm of Matias and Vishkin [9, Theorem 3], we sort these integers. On a CRCW-PRAM, this takes $O(\log n \log\log n)$ expected time and $O(n \log\log n)$ work.

Given this sorted sequence, we compute how often each value occurs. Then, we can compute $\Sigma(S, R)$. This takes $O(\log n)$ time and $O(n)$ work.

Lemma 10 *Let R be the set of hypercubes that are computed by the above algorithm. Then, R is a δ-covering of S. Moreover, the algorithm computes R and $\Sigma(S, R)$ in $O(\log n \log\log n)$ expected time and $O(n \log\log n)$ work.*

Proof: The proof follows from the above discussion. ∎

Theorem 3 *There exists a CRCW-PRAM algorithm that computes a real number δ such that $k \leq r_\infty(\delta) \leq 80^D(k + 3n)$. The algorithm takes $O((\log n)^2 \log\log n)$ expected time and $O(n \log n \log\log n)$ work.*

Proof: The correctness of the algorithm follows in the same way as in Theorem 1. The initialization of the algorithm takes $O((\log n)^2)$ time and $O(n \log n)$ work. Combining this with Lemmas 9 and 10, the complexity bounds follow. ∎

4.2 The enumeration phase

The parallelization of the enumeration phase follows easily form the sequential algorithm of Subsection 3.2. In Step 1—construction of the $D\delta$-covering R of S—we use Steps 1-3 of the procedure in the previous subsection. This gives for each point p a vector of length D indicating the lower-left corner of the δ-cube of R that contains p. Then, we use any optimal sorting algorithm to sort these cubes, more precisely, the lower-left corners of these cubes. Hence, Step 1 of the enumeration phase takes $O((\log n)^2)$ time and $O(n \log n)$ work.

The sorted sequence of δ-cubes allows us to do point location in logarithmic time. (Note that we basically apply the slab method for point location.) In Step 2, we find the same $O(k+n)$ pairs as in Step 2 of Subsection 3.2. This takes $O((\log n)^2 \log\log n)$ time and $O(n \log n + k)$ work.

For Step 3, we use Cole's optimal algorithm of [3] that finds the k-th smallest element in the list of $O(k+n)$ L_t-distances. This algorithm takes $O((\log n)^2 \log\log n)$ time and $O(n+k)$ work. Given this k-th smallest distance d_t^k, we select all k L_t-distances that are at most equal to d_t^k. This also takes $O((\log n)^2 \log\log n)$ time and $O(n+k)$ work.

Lemma 11 *Given the real number δ of Theorem 3, we can find the k closest pairs in the set S in $O((\log n)^2 \log\log n)$ time and $O(n \log n + k)$ work.*

Proof: The correctness proof is the same as in Lemma 7. The complexity bounds follow from the description just given. ∎

This concludes the description of our parallel k closest pairs algorithm. We combine Theorem 3 and Lemma 11 to obtain the following result.

Theorem 4 *Let S be a set of n points in D-space and let $1 \leq k \leq \binom{n}{2}$. There exists a CRCW-PRAM algorithm that finds the k closest pairs (w.r.t. the L_t-metric) in the set S in $O((\log n)^2 \log\log n)$ expected time and $O(n \log n \log\log n + k)$ work.*

Of course, we can use the algorithm with $k = 1$ to find the closest pair in the set S. For dimensions that are greater than two, this gives the currently best result:

Corollary 3 *Let S be a set of n points in D-space. There exists a CRCW-PRAM algorithm that finds the closest pair (w.r.t. the L_t-metric) in S in $O((\log n)^2 \log\log n)$ expected time and $O(n \log n \log\log n)$ work.*

5 Concluding remarks

We have given an optimal sequential algorithm for the k closest pairs problem. As mentioned already, the constants that appear are rather high. They are valid, however, for any L_t-metric, $1 \leq t \leq \infty$. By a more careful analysis, the constants can be improved. In particular, by taking a specific t, e.g. $t = 2$, in which case we consider the Euclidean metric, it is easy to improve them. They remain, however, of the form $c^{D \log D}$.

There remain some interesting problems that need more attention. Our algorithm approximates the L_∞-distance with a certain rank r. Can it be modified to find the exact L_∞-distance with rank r? Note that Salowe [10] solves this problem in $O(n(\log n)^D)$ time. Another problem is to find the L_t-distance with rank r for other values of t.

The algorithm presented here finds the k smallest distances, but it does not output this sequence in sorted order. Of course, we can solve the "sorted k closest pairs problem" in $O((n+k)\log n)$ time. It is an open problem if the time complexity can be improved to $O(n\log n + k)$. In particular, it is an open problem if we can sort all $\binom{n}{2}$ distances in $O(n^2)$ time.

Finally, can we use the techniques of this paper to improve the time bounds in [7] for the k furthest pairs problem, or for the k closest/furthest bichromatic pairs problem? (Note that the results in [7] only hold for the planar case.)

For the parallel version of the algorithm, the product of the time complexity and the number of processors is $O(n\log n \log\log n + k)$. It would be interesting to remove the $\log\log n$-factor from this product. Note that this factor comes from integer sorting. Hence, any improvement for this problem leads to an improvement of our algorithm. Moreover, it would be interesting to remove randomization from this algorithm. Finally, it is an open problem to devise an optimal parallel algorithm for the higher-dimensional closest pair problem, i.e., for the case $k = 1$.

Acknowledgement

The authors thank Torben Hagerup and Christine Rüb for their help with the parallel version of the algorithm.

References

[1] P.K. Agarwal, B. Aronov, M. Sharir and S. Suri. *Selecting distances in the plane.* Proc. 6-th ACM Symp. on Comp. Geom., 1990, pp. 321-331.

[2] J.L. Bentley and M.I. Shamos. *Divide-and-conquer in multidimensional space.* Proc. 8-th Annual ACM Symp. on Theory of Computing, 1976, pp. 220-230.

[3] R. Cole. *An optimally efficient selection algorithm.* Information Processing Letters **26** (1988), pp. 295-299.

[4] R. Cole and M.T. Goodrich. *Optimal parallel algorithms for point-set and polygon problems.* Algorithmica **7** (1992), pp. 3-23.

[5] M.T. Dickerson and R.S. Drysdale. *Enumerating k distances for n points in the plane.* Proc. 7-th ACM Symp. on Comp. Geom., 1991, pp. 234-238.

[6] D.B. Johnson and T. Mizoguchi. *Selecting the Kth element in $X+Y$ and $X_1 + X_2 + \cdots + X_m$.* SIAM J. Comput. **7** (1978), pp. 147-153.

[7] N. Katoh and K. Iwano. *Finding k farthest pairs and k closest/farthest bichromatic pairs for points in the plane.* Proc. 8-th ACM Symp. on Comp. Geom., 1992, pp. 320-329.

[8] H.P. Lenhof and M. Smid. *The k closest pairs problem.* Unpublished manuscript, 1992.

[9] Y. Matias and U. Vishkin. *On parallel hashing and integer sorting.* J. of Algorithms **12** (1991), pp. 573-606.

[10] J.S. Salowe. *L-infinity interdistance selection by parametric search.* Inform. Proc. Lett. **30** (1989), pp. 9-14.

[11] J.S. Salowe. *Enumerating interdistances in space.* Internat. J. Comput. Geom. Appl. **2** (1992), pp. 49-59.

[12] C. Schwarz, M. Smid and J. Snoeyink. *An optimal algorithm for the on-line closest pair problem.* Proc. 8-th ACM Symp. on Comp. Geom., 1992, pp. 330-336.

[13] M.I. Shamos and D. Hoey. *Closest-points problems.* Proc. 16-th Annual IEEE Symp. on Foundations of Computer Science, 1975, pp. 151-162.

[14] M. Smid. *Maintaining the minimal distance of a point set in less than linear time.* Algorithms Review **2** (1991), pp. 33-44.

[15] P.M. Vaidya. *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem.* Discrete Comput. Geom. **4** (1989), pp. 101-115.